11-5-1993

Vertex Ordering for a Partitioning-based Fitting Algorithm for an EPLD Device

Tongjun Gao
Portland State University

AN ABSTRACT OF THE THESIS OF Tongjun Gao for the Master of Science in Electrical and Computer Engineering presented November 5,1993.

Title: Vertex Ordering For A Partitioning-Based Fitting Algorithm For An EPLD Device

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Malgorzata Chrzanowska-Jeske, Chair

[redacted]

Marek A. Perkowski

[redacted]

Maria E. Balogh

As the Application-Specific Integrated Circuit(ASIC) technology develops to the trend of high density and modulization, the ASIC device market has been dominated gradually by the more complex Erasable Programmable Logic Devices (EPLDs) and the Field Programmable Gate Array(FPGAs) instead of the ordinally Programmable Logic Devices(PLDs). Meanwhile, the design automation system for such programmable devices has also moved from schematic entry design to high level hardware description language entry design. Usually, the whole design automation process

consists of three phrases, the high level hardware description language compiler, the logic synthesis stage and the layout synthesis stage. Though the layout synthesis stage contains placement and routing, for some highly restricted connection architecture devices, placement and routing have to be considered together as a fitting problem. This thesis concentrated on the utilization of the Heuristic methods, which can be described as vertex ordering and global vertices number estimation, on an Architecture-Driven Partitioning fitting algorithm. The test results showed that the heuristic algorithm can beat the comparable algorithm in several fields. These prove the correctness of our heuristic methods and they can be used to guide the future work on the fitting problem of other similar programmable devices.

VERTEX ORDERING FOR A PARTITIONING-BASED FITTING ALGORITHM FOR AN EPLD DEVICE

by

TONGJUN GAO

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1993

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Tongjun Gao presented November 5, 1993.

[redacted]

Malgorzata Chrzanowska-Jeske, Chair

[redacted]

Marek A. Perkowski

[redacted]

Maria E. Balogh

APPROVED:

[redacted]

Rolf Schaumann, Chair, Department of Electrical Engineering

[redacted]

Roy W. Koch, Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENTS

I am very thankful to my parents, my uncle and my sister. They gave me strong support and encouragement during the period of my graduate study in the United States. Their love created my energy to work and study hard.

I would like to thank Dr. Malgorzata Chrzanowska-Jeske, my advisor, Chair of the Committee, for her brilliant guidance and patient encouragement throughout the research and the preparation of the thesis.

I would like to thank other Committee members: Dr. Marek A. Perkowski and Dr. Maria E. Balogh for their assistance on the thesis.

Thanks to my friends and the faculty and staff of the Electrical Engineering Department for their friendly support and encouragement in many ways.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Recently, as the Application-Specific Integrated Circuit(ASIC) technology develops towards high density architecture design, the programmable logic devices(PLDs) have gradually been substituted by the more complex Erasable Programmable Logic Devices (EPLDs). Furthermore, the Field Programmable Gate Arrays (FPGAs), which combine the design characteristics of both high density architecture and modular function blocks, have also emerged as the new generation device in the current ASIC market. Meanwhile, the design automation system has also moved from schematic entry design to high level hardware description language entry design.

Currently, the IEEE standard VHSIC Hardware Description Language(VHDL) has been used in many design automation system [1]. The VHDL description of the entire design, which is entered by the designer, is translated to a device programming file. Such a device programming file is the internal format expressions that can separately describe the sequential and combinational logic of the entry design circuit [2]. Then the compiler's logic synthesis step divides the whole circuit into pieces of small logic, each of which can be realized as a single module. The result of the logic synthesis [3] can be represented by a netlist that contains all the modules of the design circuit and the connections between these modules. Each small module of the design circuit can be mapped into a single macrocell of the selected device [4], [5], [6]. Finally, the layout synthesis step assigns each module of the netlist to a physical macrocell of the device and uses the proper available wire segments from the routing resources of the device to realize the

connections between the macrocells [7], [8], [9], [10], [11], [12].

For some complex programming logic devices, the connections between the macrocells are so restricted that the placement of modules have to take the routing restriction into account during the whole layout synthesis' process. Therefore, the layout synthesis problem on those highly restricted connection architecture EPLDs/FPGAs needs to be formulated as a fitting problem, which combines the placement and routing together [13], [14], [15]. [16]. The sub-system for solving the fitting problem of the whole design automation system is called fitter.

This thesis concentrates on the fitting problem of one of the Application-Specific State Machine Devices(ASSMDS), the CY7C361 EPLD, which was introduced by Cypress Semiconductor. The fitting problem can be presented as mapping the netlist obtained from high-level synthesis stage into the chip's physical resources [13], [14]. The netlist, which represents the realization of the design circuit, is represented as a directed cyclic graph. The physical resource of the device architecture, which is the realization of the internal macrocells and the connections between them, can also be represented as a directed cyclic graph. The fitting problem was formulated as the labeled graph isomorphism between the netlist graph and the sub-graph of the resource's graph [15]. However, due to the strongly limited connectivity of the new EPLD/FPGA devices, the fitting problem has to be generalized as a graph monomorphism problem [17] with some additional mapping constraints [13], [14].

A vertex shifting arrangement mapping algorithm has been developed be Cypress Semiconductor to solve the CY7C361 fitting problem [16]. This approach tried to map the symbolic vertices of the netlist into the physical locations of the device by shifting the netlist vertex order. Due to the local approach of this algorithm, the search time was excessive. By considering the device architecture, an Architecture-driven partitioning fitting algorithm (PABFIT) has been developed to improve the speed but at the same time

to maintain the exactness of the fitting problem. [13], [14], [15]. However, PABFIT still experienced the same problem for some large circuit designs. In this thesis we will present a new vertex ordering heuristic which will be used in the original Architecture-driven partitioning fitting algorithm (PABFIT) to improve the algorithm performance. It is shown on the attached examples that the partitioning-based fitting algorithm with heuristic vertex ordering (PABFIT.h) can find a feasible solution much faster than the original PABFIT algorithm.

Chapter II presents EPLD devices, concentrating on the CY7C361 architecture and its partitioning properties. Chapter III describes PABFIT-the Partitioning Based fitter. Chapter IV and Chapter V presents the mathematic formulation on the new Heuristic and the Algorithm of the Heuristic Methods. Chapter VI give the evaluation of the results. Chapter VII presents the complexity analysis. Chapter VIII summarizes our work and address the future work.

CHAPTER II

EPLD DEVICES

PLDs use a number of different programming technologies. The ultraviolet erasable PLDs, commonly called EPLDs, are based on the CMOS technology. These EPLD devices have the reprogrammable ability. Though this technology won't normally affect the use of the devices, it can affect how easily the devices can be programmed. CMOS technology can be easily tested and used to design high density device.

II.1 RESTRICTED-CONNECTIVITY EPLD DEVICES

New EPLD devices have recently been developed by using different architecture from the previous PLDs in order to gain the goal of complexity. One developing trends is modulization. The device is designed with configurable outputs, which are enhanced with special circuitry and are called output macrocells. The device has a number of identical output macrocells. The control fuses for the macrocells allow each macrocell to be configured in one of several basic configurations. The macrocells can be programmed as AND, OR gates, Multiplexors, D flip-flops, or other more complex gates depending on the chip architecture.

Another developing trend is segmented architecture. As PLDs increase in complexity, the size of the programmable array quickly becomes unmanageable. For this reason, some manufacturers have developed device architecture that are segmented into smaller arrays with limited interconnections. The device's programmable array splits into several

identical parts that have limited interconnects. This segmentation means that designs being implemented in such device must be partitioned and allocated to the parts based on the amount of the required interconnection. The segmentation of the interconnection array results in a highly restricted connectivity nature among the macrocells.

The CY7C361 chip, an Application Specific State Machine Device, introduced by Cypress Semiconductor, is one of these new architecture EPLD/FPGA. It combines the characteristics of both modulation and segmentation.

II.2 CYPRESS EPLD

II.2.1 CY7C361 Chip Architecture

The CY7C361 chip contains 32 internal macrocells [18]. These 32 physical macrocells are positioned in one column, and numbered from 1 to 32 from the top to the bottom of the column. These 32 macrocells are also called state macrocells.

By using C_IN chain inputs, all state macrocells can be configured as shift registers. This C_IN chain input is a very short hard-wired connection between the adjacent macrocells and can be regarded as a condition reset to a configurable macrocell.

The excitation functions and the reset functions of the flip-flops in the state macrocells are designed as unique high speed gates called Condition Decodes (CDEC). CDEC gate is an AND of two fan-in unlimited gates, AND and NAND. These CDEC gates take in primary inputs and feed back the state macrocell outputs to other CDEC gate. Therefore, one CDEC gate can take the primary inputs from the external pins and the feed back argument signals from another internal CDEC gate.

The CY7C361 architecture contains 32 state macrocells, 8 local resets and 1 global reset. The whole architecture is partitioned into two groups of 16 state macrocells with

their associated reset signals and CDEC planes. Each of these 16-macrocell groups is partitioned again into two groups of 8 state macrocells. Each group of 8 macrocells consists of two groups of 4 macrocells with a separate local reset CDEC. Each group of 4 macrocells is considered as one physical local reset group (LRG). In this 4-macrocell LRG, the outputs of the first two macrocells are available to all macrocells in the subgroup of 8 macrocells, output of the third one to all macrocells in subgroup of 16 macrocells, and output of the fourth one is global to all 32 macrocells. The outputs from all macrocells in each of the 8-cell groups are available at the inputs of all the macrocells belonging to that group. According to the output availability, the macrocells of CY7C361 can be classified into three different groups: (1) global, (2) intermediate, and (3) local [13], [14], [16]. The description of these three groups will be presented later.

A state macrocell can have input signals from Condition, Global Reset, and Local Reset. These condition and reset logic are realized by the CDEC gates as well. In addition, C_IN chain input is also available to every state macrocell as a separate condition excitation signal. For each physical Local Reset Group of four cells, there exits one separate Local Reset signal, but there is only one Global Reset signal for all state cells. In CY7C361 architecture, the 32 internal macrocells are divided into 8 local reset groups(LRGs), each of which contains 4 cells. These 8 physical local reset groups are named LRG_1, LRG_2,..... LRG_8. Each physical local reset group LRG_i is associated with exactly one physical local reset plr_i. Microcells triggered by physical local resets are called TOGGLE cells. All TOGGLE cells in the same physical LRG_i must be triggered by the same local reset plr_i. In each physical local reset group, there exists two local, one intermediate, and one global macrocells [13], [14], [16].

II.2.2 Partitioning Property

As we have mentioned before, the fitting algorithm tries to assign a netlist of the circuit, which is obtained from the logic synthesis stage, to the physical resource of the chip. According to the architecture of the CY7C361 chip, we know that the internal macrocells of the device can be partitioned into three groups, global, intermediate and local, and the macrocells in one group can have their own distinctive properties, the connectivity and the connection constraints, which are different from those in the other groups. Since the netlist should have the same partitioning properties as the routing domain of the physical device, a two-level partitioning fitting algorithm has been developed to solve the fitting problem of the CY7C361 chip. Before describing the fitting algorithm, we will first explain the architecture constrains of the CY7C361 and its corresponding partitioning properties.

The physical resource of the CY7C361 chip, the 32 macrocells, the condition C_IN chain reset signals, the global and local reset signals, and the possible connections between them are represented by the directed physical graph $G_p = (V_p , E_p)$, denoted Physical Connectivity Graph. It consists of the vertex set V_p ={ v_i | v_i represents macrocell } and the edge set E_p ={ e_i | e_i represents the possible connection between V_i and V_j} [13], [14], [18].

The Physical Connectivity Graph G_p (V_p , E_p) can also be represented by an Interconnection Matrix or Adjacency Matrix, which is shown in Figure 1. The first 32 rows and 32 columns, P_1, P_2 , ... P_{32}, represent the state macrocells. The first 32 rows in the Adjacency Matrix represent the outputs of the macrocells, and the first 32 columns represent the inputs to these macrocells. If the physical connection exists between the output of the row cell and the input to the column cell of the resources' graph, it is indicated by the symbol "|". If no physical connection exists, just keep the empty space in that position. A chain of down arrows is listed on the left hand side of the graph to

indicate the C_IN chain connection. C_IN chains have to be mapped in the specified order to the adjacent macrocells P_i and P_{i+1}. The row/column 33, indicated by P_{33}, gives the outputs and inputs of the global reset, and the row 34 to row 41/column 34 to column 41, indicated by $P_{34} - P_{41}$, gives the available inputs/outputs of the local resets [13], [14].

The Interconnection Matrix given in Figure 1 is ordered according to the real physical location of the macrocells on the chip. To observe the special connectivity symmetry of this architecture, the Interconnection Matrix is reorganized and is presented in Figure 2 [13], [14]. The global reset is not included in the Adjacency Matrix given in Figure 2 because the output of the global reset can be available to the inputs of all the state macrocells which means no restrictions on its output connectivity. Additionally, the connections of the local resets exhibit the same partitioning property as the Adjacency Matrix of the macrocells shown in Figure 2. Therefore, local reset vertices given by rows/columns p34-p41 of the matrix in Figure 1 are not included.

Before describing the partitioning properties of the adjacency matrix in Figure 2, the classification of the macrocells will be described.

According to the connection availability of the output of a state macrocell $p_i \in \{p_1, \ldots, p_{32}\}$ to the input of state macrocell $p_j \in \{p_1, \ldots, p_{32}\}$ $(i \neq j)$, macrocell p_i can be defined as one of the following types:

Type I: *Global macrocell:* the macrocell p_i is considered global if its output is available at all other 32 macrocells.

Type II: *Intermediate macrocell:* the macrocell p_i is considered intermediate if its output is available at 16 other macrocells, belonging to the same 16-macrocell block.

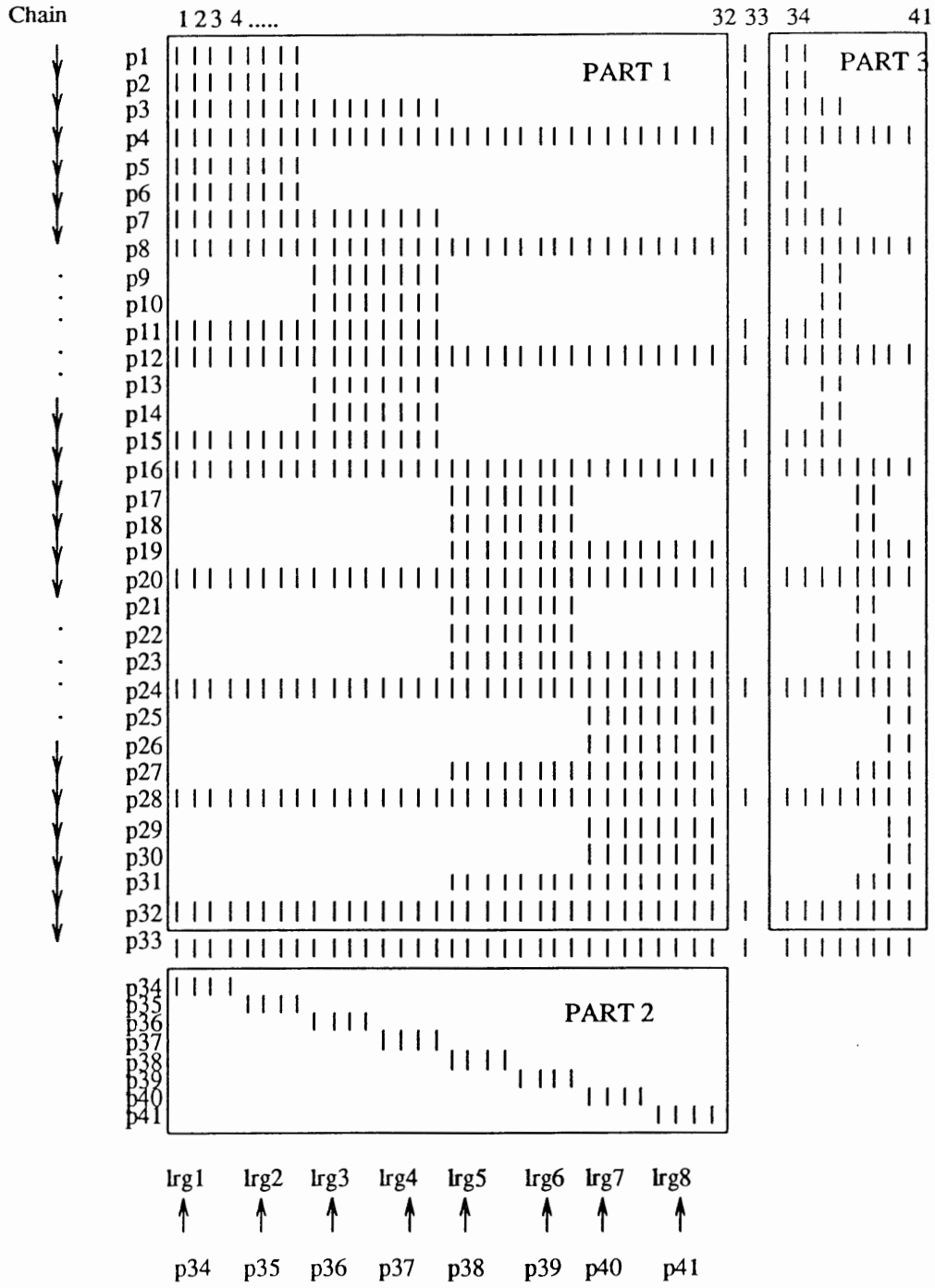


Figure 1. Interconnection Matrix of CY7C361.

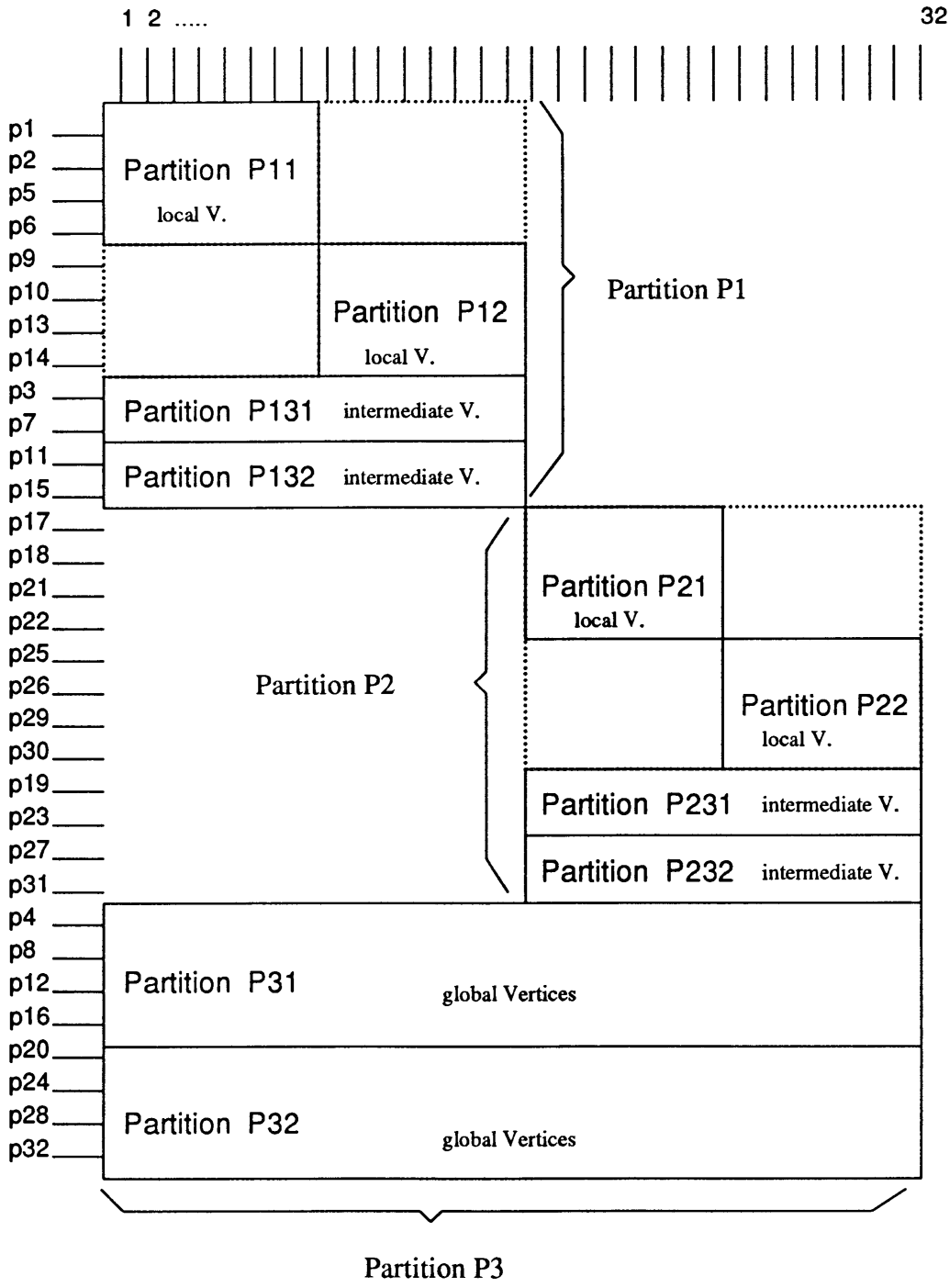


Figure 2. Partitioning Properties of the Adjacency Matrix.

Type III: *Local macrocell:* the macrocell p_i is considered local if its output is available at 8 other macrocells, belonging the same 8-macrocell block.

Therefore, the CY7C361 chip contains 8 global, 8 intermediate and 16 local macrocells, which can be seen in the Adjacency Matrix in Figure 2. As it was shown before, the first level partitioning separates the macrocells into 3 groups: P_1, P_2 and P_3. The partitions P_1 and P_2 contain the intermediate and the local macrocells. The global macrocells are in partition P_3 and have their outputs available at all other macrocells in P_3 and at all macrocells in P_1 and P_2. All macrocells in partition P_1 have outputs available only at macrocells in the same partition P_1 and in P_{31}. Analogously, the macrocells in partition P_2 have outputs available only at macrocells in P_2 and in P_{32} [16].

The partitioning of P_1 and P_2 is identical. Therefore, only the partitioning of P_1 is explained in the following. The second level partitioning separates P_1 into P_{11}, P_{12}, P_{13}. The partitions P_{11} and P_{12} contain only the local macrocells. The intermediate macrocells are in partition P_{13} and have outputs available at all other macrocells in P_{13} as well as at all macrocells in P_{11} or P_{12}. The local macrocells of partition P_{11} have outputs available only at macrocells in the partition P_{11} and in P_{131}. Analogously, the macrocells in partition P_{12} have outputs available at macrocells in P_{12} and in P_{132}.

It can be observed in Figure 2 that the availability of the outputs of the state macrocells at the inputs to the CDECs of the local reset signals exhibits the same partitioning properties as the state macrocell connections. The state macrocell which is global in respect to the other macrocells is also global in respect to the local reset connections. The same is true for intermeditate and local macrocells.

Based on the above description of the chip architecture, three groups of constraints were distinguished: state macrocell connectivity constraints, global reset constraints, and local reset constraints [13], [14], [16].

CHAPTER III

PABFIT- PARTITIONING-BASED FITTER

III.1 FITTING PROBLEM FORMULATION

We are going to solve the problem of mapping the netlist obtained from the logic synthesis stage into the physical resources of the CY7C361 device, from Cypress. The netlist of the design is represented by the directed Symbolic Graph $G_s = (V_s , E_s)$. The vertices $v_i \in V_s$ represent the macrocells, and the connections between the macrocells are represented by the edges $e_i \in E_s$. The local resets constraints are not considered in the partitioning phase. The formulation of the fitting problem was stated as follows:

Problem Formulation: Given a directed physical connectivity graph $G_p = (V_p , E_p)$ and a directed symbolic graph $G_s = (V_s , E_s)$, determine if the symbolic graph G_s is monomorphic to the physical connectivity graph G_p, and if so, find the mapping (monomorphism Π) of the symbolic vertices $v_{s_i} \in V_s$ to the physical vertices $v_{s_i} \in V_p$, so that no global and local reset restriction are violated [16].

The graph monomorphism problem [17] is of high complexity. To reduce this complexity we take advantage of the partitioning properties of the physical connectivity graph G_p [13], [14]. A symbolic graph G_s must exhibit the same partitioning properties as the physical connectivity graph G_p. A two-level partitioning is performed on the symbolic graph G_s, subject to the first- and second-level partitioning properties of the physical connectivity graph G_p. The final assignment of symbolic vertices to the physical locations (state macrocells and reset cells) is performed in the Physical Placement stage [13],

[16].

III.2 THE PABFIT ALGORITHM

The partitioning-based fitting algorithm(PABFIT) is divided into two parts: The First- & Second-level Partitioning and the Physical Placement. The Partitioning part consists of two stages: P3 Assignment, First Partition; P13,P23 Assignment and Second Partition. The flowchart of the PABFIT is shown in Figure 3.

In the P3 Assignment stage, the set of vertices, from the symbolic graph G_s, is chosen randomly to be assigned as the global vertices. The chosen vertices are placed in the block P_3. The remaining vertices are processed by the First Partition and divided into two blocks, block P_1 and block P_2. There are no connections between blocks P_1 and P_2. So the first stage of the Partitioning part of the PABFIT separated the input vertices into partition P_1, partition P_2 and partition P_3 [13], [14], [15].

In the second stage of the Partitioning part, the partitions P_1 and P_2 themselves are partitioned further in a similar way. The P_{13} - P_{23} Assignment selects the Intermediate macrocells from the block P_1 or P_2 and places them into the blocks P_{13} or P_{23} respectively. The Second Partition deals with the remaining vertices in blocks P_1 or P_2 and tries to divide blocks P_1 or P_2 into two sub-blocks P_{11} and P_{12} or P_{21} and P_{22} respectively. The vertices in subpartition P_{11} and P_{21} and the vertices in subpartition P_{12} and P_{22} have no output connections to each other.

After the First- & Second-level Partitioning, each vertex $v_{s_i} \in V_s$ is assigned to a single partition P_α. Then the Physical Placement performs a one-to-one mapping of each vertex v_{s_i} to a single physical macrocell p_j of the device. If no connectivity and reset restrictions are violated, PABFIT has found one feasible solution to the fitting problem [13], [14], [15].

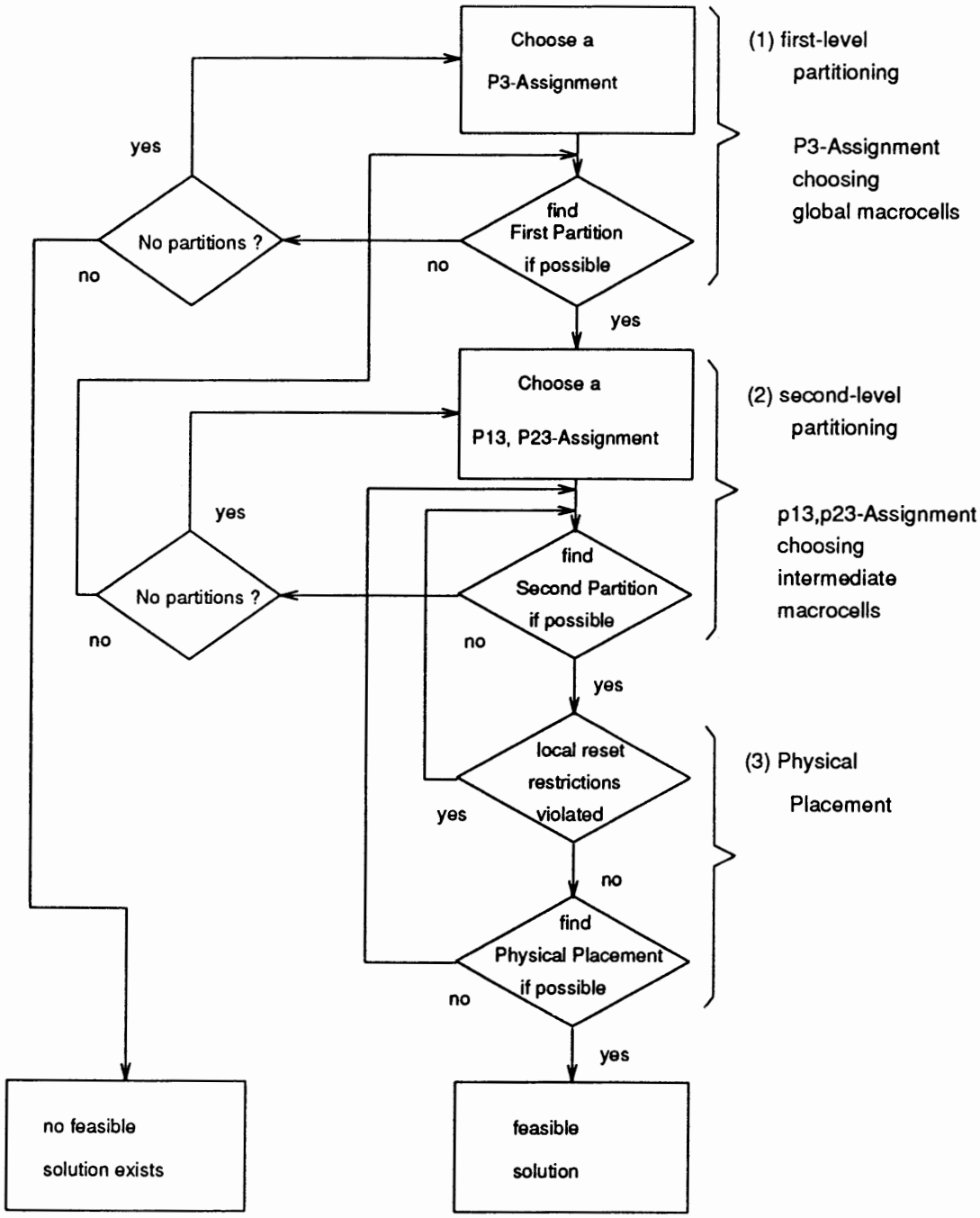


Figure 3. Flowchart of the PABFIT Algorithm.

III.3 DRAWBACKS OF THE PABFIT ALGORITHM

In this work, we will concentrate on the first step of the PABFIT: the P3 Assignment, because the P3 Assignment has an important effect on the executing time of the algorithm. Therefore, we will use heuristic approach to control the assignment process. The P3 Assignment is the assignment of vertices $v_{Si} \in V_s$ to the partition P_3. The outputs of the macrocells mapped to the vertices assigned to P_3 are available at the inputs to all other state macrocells and reset cells, because the physical macrocells in P_3 are global macrocells. All other vertices that are not assigned to P_3 can be assigned only to intermediate or local macrocells (partitions P_1 , P_2 and their subpartitions). After the P_3 Assignment, the output edges of vertices $v_{Si} \in V_s$ that are assigned to P_3 are removed. The disconnected components G_{Ci} of the symbolic graph G_s are assigned to different partitions during the First Partitioning stage. By properly choosing the vertices assigned to P_3, we should be able to find a solution, if one exists, much faster, and without going through many possible P_3 assignments. As a result, a good P_3 Assignment can speed up the search for a feasible solution and reduce the CPU time.

Basically, there are two aspects of the P_3 Assignment which can influence the speed of the whole algorithm.

- If we know that the minimum number of global vertices is k ($k \geq 0$), then the possible number of the global vertices (number given by m) can be assigned from k to 8. So the order of how many vertices (number given by m) out of all the input vertices are assigned to P_3 can be changed. It can start with $m = 0$ and increase to $m = 8$ step by step, or it might start with $m = 7$, continue to $m = 8$, and than back to $m = 0$, until all possible cases are explored.

- The order in which m vertices out of all possible vertices are assigned to P_3 can also be changed. For example, it can start with assigning vertex

v_1 , v_2 , and v_3 (m=3) to P_3, or it might start with v_2 , v_3 , and v_4. Then it continues assignment until all possible combinations of m vertices are explored.

Therefore, heuristics can be applied to control the P_3 Assignment in order to speed up the search for a feasible solution [19]. The heuristic approach depends highly on certain common properties of the input vertices. We have tested PABFIT on a large number of examples to find out whether the vertices assigned into P_3 in the final feasible solution have some common properties. According to such common properties, we can develop new heuristic approaches such as:

- Choose the proper order of how many vertices (number given by m) out of all the input vertices are assigned to P_3.
- Chose the proper order of how these m different vertices are assigned to P_3.

Assume that all possible P_3 assignments determined by PABFIT form a searching space. The developed heuristic changes the starting position of the search and the searching order inside the searching space. No possible P_3 assignments are excluded, and PABFIT can still explore all possible P_3 Assignments. However, with good heuristic, it is more likely that a good P_3 assignment can be found sonner and therefore a feasible solution can be also identified [19], [20].

The original PABFIT [16] uses only one heuristic indicator in the P_3 Assignment stage. That indicator is based on a high In-Degree of some vertices of the symbolic graph compared to the other vertices. Therefore, the first step is to determine the In-Degree of all the vertices of the symbolic graph and store all the values in a so-called degree vector. Then, the degree vector is ordered according to the increasing degree and the difference Δ between adjacent elements of the degree vector is calculated. The maximum difference is denoted by Δ_{max} and the element of the degree vector where Δ_{max} occurs is denoted by $i_{\Delta_{max}}$. Figure 4 here shows an example of the degree vector,

Δ_{max} , and $i_{\Delta_{max}}$.

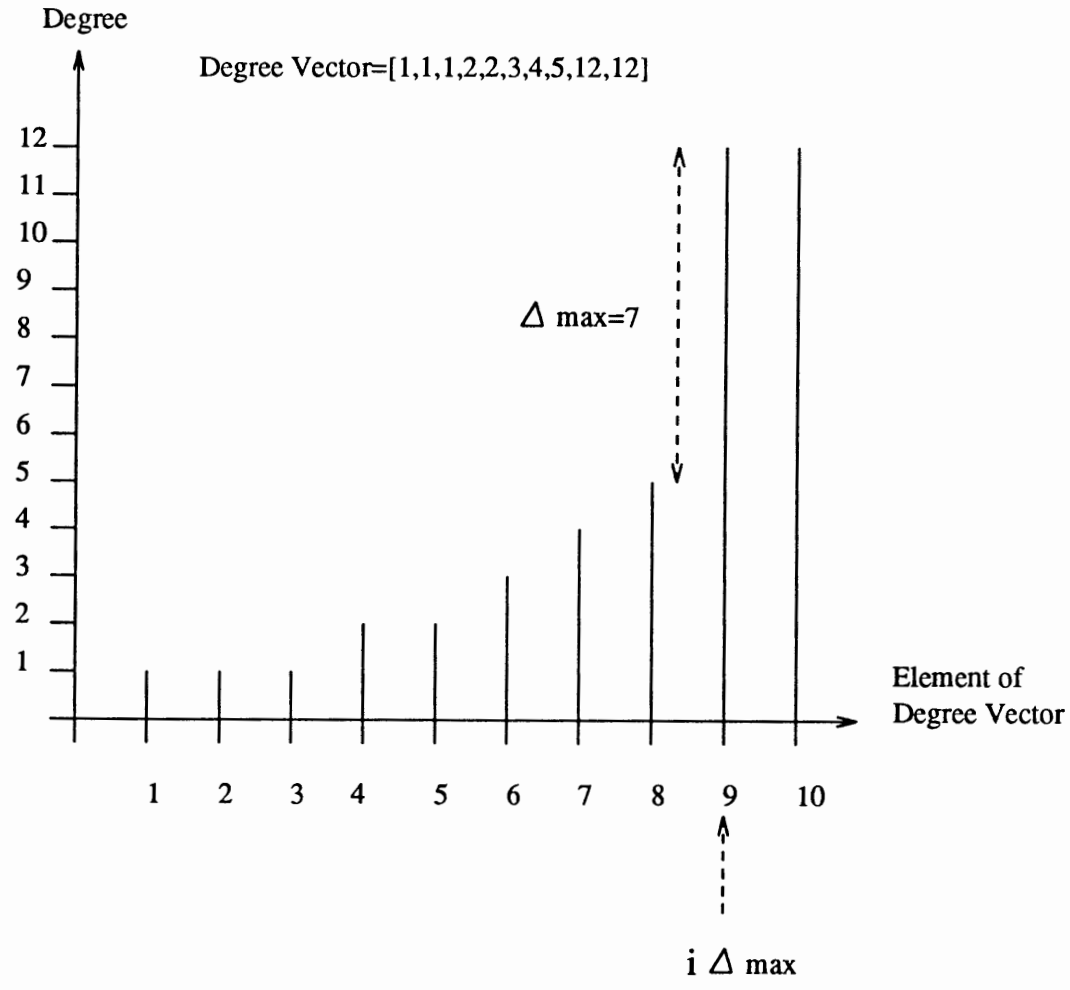


Figure 4. High In-Degree Vector.

If $\Delta_{max} > 5$, the algorithm starts assigning $m = \Delta_{max}$ vertices to the partition P_3 and increases the value of m step by step after all possible combinations of have been explored. The variable m is increased until it reaches $m = 8$, then the PABFIT starts again at m=0 and counts up to $m = \Delta_{max}$. If $\Delta_{max} > 8$, PABFIT starts with m=8.

The comparison between the PABFIT without heuristic and with High_In_Degree heuristic, on a set of examples, is presented in Table I.

TABLE I

IMPROVEMENT DUE TO HIGH_IN_DEGREE HEURISTIC

name	max in_ degree	Diff_max	M_start	CPU time heuristic [s]	CPU time non_heuristic [s]
cadman_b	9	6	6	58.3	150.1
cadman_ba	9	6	6	136.6	226.9
tsrbug_b	14	13	8	0.1	5h

For all the examples where PABFIT applies High_In_Degree heuristics the CPU time is decreased compared to the non-heuristic PABFIT, since the heuristic depends highly on certain properties of the input netlist. For some examples with such properties, heuristic would speed up the search, but for others without such properties, the CPU time was even increased. Especially for some examples with a large searching space, PABFIT would keep on running for a long long time without finding out the feasible solution. Therefore, more general heuristics should be developed to improve PABFIT performance on average set of examples.

CHAPTER IV

MATHEMATICAL FORMULATION OF THE VERTEX ORDERING

It is necessary to analyze the architecture of the chip and the general properties of the input netlist to develop some new approach which can be applied in PABFIT. First, let's analyze the characteristics of the netlist which will affect the searching order of the P_3 assignment. Then we will explain the heuristic methods which will choose the number of vertices assigned to the P_3 assignment, or the number of possible global macrocells.

IV.1 SEARCHING ORDER OF P_3 ASSIGNMENT

IV.1.1 Outstanding Vertices

Before introducing the term *outstanding vertices,* let's explain the total degree and the multiple degree of the vertices on examples shown in Figure 5. If vertex V_1 is related to vertex V_3 through vertex V_s, then vertex V_s provides an opportunity for vertex V_1 to be related to vertex V_3. Let's represent it as $V_1 \rightarrow V_s \rightarrow V_3$, obviously, it is the same as $V_1 \rightarrow V_s \rightarrow V_4$, $V_1 \rightarrow V_s \rightarrow V_5$, $V_2 \rightarrow V_s \rightarrow V_3$, $V_2 \rightarrow V_s \rightarrow V_4$, and $V_2 \rightarrow V_s \rightarrow V_5$. Therefore, we can say that vertex V_s provides a total of 6 opportunities for one set of vertices (V_1 , V_2) to be related to another set of vertices (V_3 , V_4 , V_5). Actually, vertex V_s is connected with 5 other vertices V_1, V_2, V_3, V_4, V_5.

The Total Degree of one certain vertex indicates how many other vertices are adja-

cent to this certain vertex, or indicates how many other vertices are output vertices or input vertices of this certain vertex. It is a sum of In-Degree and Out-Degree of the vertex. Therefore, the Total Degree of vertex V_s determines how many other vertices can be connected to this vertex V_s. It shows vertex V_s' connectivity.

The Multiple Degree of one certain vertex indicates the possible ability (or latent capacity or potentiality) of this vertex to be adjacent to other vertices. The Multiple Degree of vertex V_s determines how many opportunities the vertex V_s can provide for the other vertices, which are connected to V_s, to be related to one another. It shows vertex V_s' ability to become a connection center. It is calculated by multiplying the vertex' In-Degree and Out-Degree.

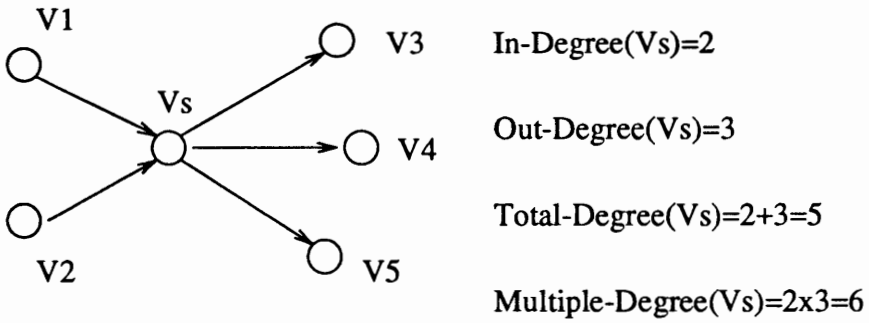


Figure 5. Vertex Degree.

According to the internal structure of the CY7C361 EPLD, which is described in detail in Chapter 2, the maximum In-Degree for all vertices is equal seventeen, the maximum Out-Degree for global vertices is equal thirty-two, for intermediate sixteen and for local eight. So, global vertices must have higher connectivity and stronger ability to become connection centers, and must have higher Multiple-Degrees and Total-Degrees. Therefore, vertices with high Multiple-Degrees and Total-Degrees can be regarded as the most possible candidates for global macrocells and they should be arranged near the top

of the vertex searching queue. We call these most possible global vertices candidates outstanding vertices.

Therefore, if we sort all the candidate vertices in the input netlist based on their total degree and multiple degree, it is likely that we can find some outstanding vertices which have higher Total Degrees and higher Multiple Degrees than the other vertices. We can anticipate that the probability of finding feasible solutions with these high degree vertices being assigned to global macrocells is very high.

Therefore, we can change the P_3 assignment vertex searching order by putting all the outstanding vertices at the beginning of the vertex searching queue. Since the P_3 assignment function poss_ab_rec() in PABFIT chooses the vertices into the P_3 assignment from the top of the searching queue, the outstanding vertices will be assigned to P_3 assignment at first. It is likely that these outstanding vertices are the real global vertices and they can be mapped into the global macrocells without violating any connection constraints. This will lead to a feasible solution very fast.

IV.1.2 Partitioning of the Input Vertices

Each state macrocell of the CY7C361 can be excited by CDEC gate outputs and C_IN chain signals. According to the input excitation signals, there are three different programmable configurations for each macrocell: (1) START, (2) TERMINATE and (3) TOGGLE.

- (1) START: When the CDEC expression is "1," the output of the START cell gate goes to "1" exactly for one clock cycle.

- (2) TERMINATE: When the C_IN signal is "1," the output of the TERMINATE cell goes to "1" in the next clock cycle, and it goes back to "0" when the CDEC expression

becomes "1."

- (3) TOGGLE: When the C_IN signal or the CDEC expression becomes "1," the output of the TOGGLE cell will toggle for the next state.

Each TOGGLE type macrocell has a programmable input connection from a local and global reset signal. The global and local resets are used to set the TOGGLE cells to a defined state, usually set to 1 by default. However, such a classification is based on the input excitation signals to the macrocell. Since the fitting algorithm has been developed based on the connection constraints between the macrocells of the physical chip, it is much more important to classify the macrocells based on their connection types than to base on their input excitation signals. Therefore, if we take the connection type between the macrocells into account, we can again classify the macrocells into three different groups: (1) Chain Cell Group, (2) Toggle Cell Group and (3) Single Cell Group.

- (1) Chain Cell Group: Each macrocell in this group belong to one and only one chain. In the chain, macrocells are connected in a line by the C_IN chain connections between every two neighbors in a chain, except the first and the last one.

- (2) Toggle Cell Group: All the macrocells in this group are TOGGLE type macrocells, and they can be triggered by the different local reset signals. They are divided into several sub-groups. In each of the separated group, all the macrocells are triggered by the same local reset signal.

- (3) Single Cell Group: All the macrocells in this group are Non-TOGGLE type macrocells without C_IN chain connections.

Since the input vertices have the same connection properties as the macrocells in the physical chip, we can also partition the input vertices into three different vertices groups: (1) Chain Group, (2) Toggle Group and (3) Single Group, similar to the connection characteristic classification of the physical macrocells. This is shown in Figure 6.

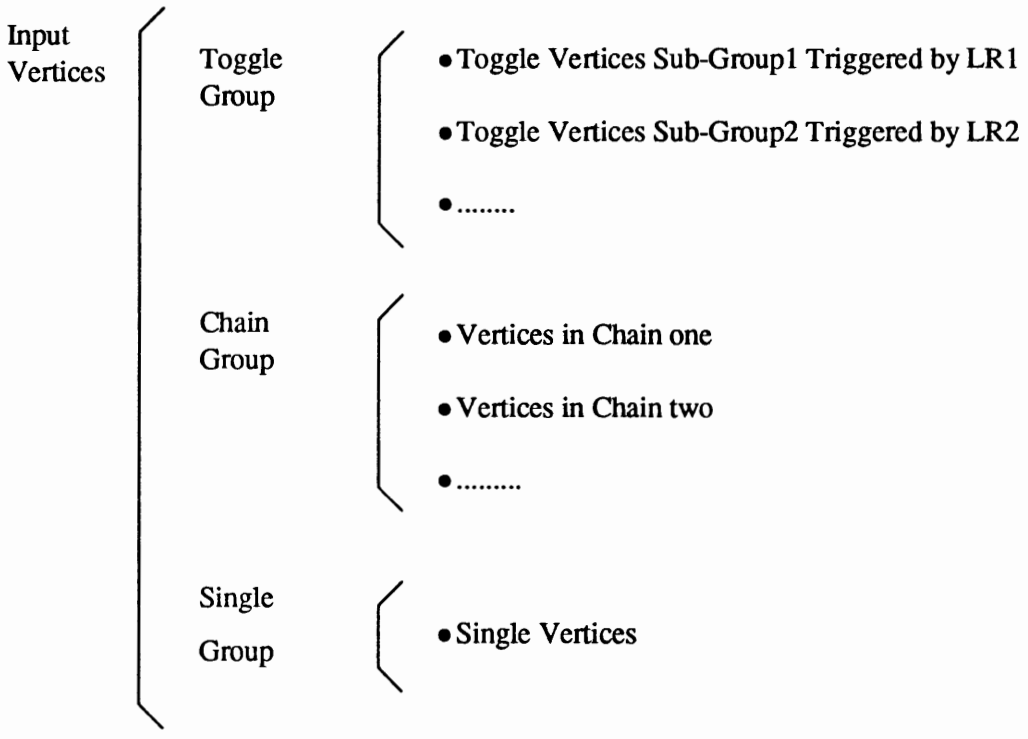


Figure 6. Partitioning of the Input Vertices.

In the P_3 assignment stage, if the vertex assigned to P_3 is from one of the chain from the Chain Group, some other vertices in the same chain have to be considered as P_3 vertices not to violate the C_IN chain connection constraints. If the P_3 vertex is from a toggle sub-group of the Toggle Group, some of the vertices in the same toggle sub-group may also be considered as P_3 vertices if the Local Reset Constraints are not violated. If it is from the Single Group, we just need to check its normal State Macrocell Connectivity Constraints. This is much simpler than the vertices from the first two groups.

Therefore, in the following sections, we only analyze the properties of vertices from the Toggle Group and the Chain Group.

IV.1.3 Properties of Toggle Group Vertices

The problems associated with the assignment of vertices from the Toggle Group will be explained on the example of "dess2_ba.fit" file, shown in Figure 7. This file represents a netlist which was synthesised by the logic synthesis stage. The Symbolic Vertices Number indicates the vertices index of the input netlist. The Type indicates the type of vertices based on the input excitation signal classification. Type 1 -> START, Type 2 -> TERMINATE, Type 3 -> TOGGLE, Type 4 -> LOCAL RESET and Type 5 -> GLOBAL RESET. Here, we only list the Type 3 TOGGLE vertices.

The function poss_ab_rec() assigns the vertices into P_3 assignment according to the top_down algorithm. In this example, the toggle vertices in several Toggle Sub-groups are at the beginning of the natural entry order of the netlist. Therefore, such toggle vertices will appear or tend to appear at the beginning of the vertex searching queue.

Therefore, it is easy for the function poss_ab_rec() to choose a highly probable P_3 assignment, which contains all toggle group vertices from the same Toggle Sub-Group, at the beginning of the search process. The determined P_3 assignment violates the local reset constraints and we call this problem of the P_3 assignment an "split-local-reset-group" problem, shown in Figure 8. Since almost all the toggle group vertices are at the beginning of the natural entry order of the netlist, such problem assignment will tend to happen at the beginning , or quite near the beginning of the search process.

As shown in Figure 7, the example netlist contains two toggle sub-groups. One toggle sub-group consists of 12 Toggle vertices, which all are triggered by the symbolic local reset lr1. Another one consists of 4 Toggle vertices, which are all triggered by the

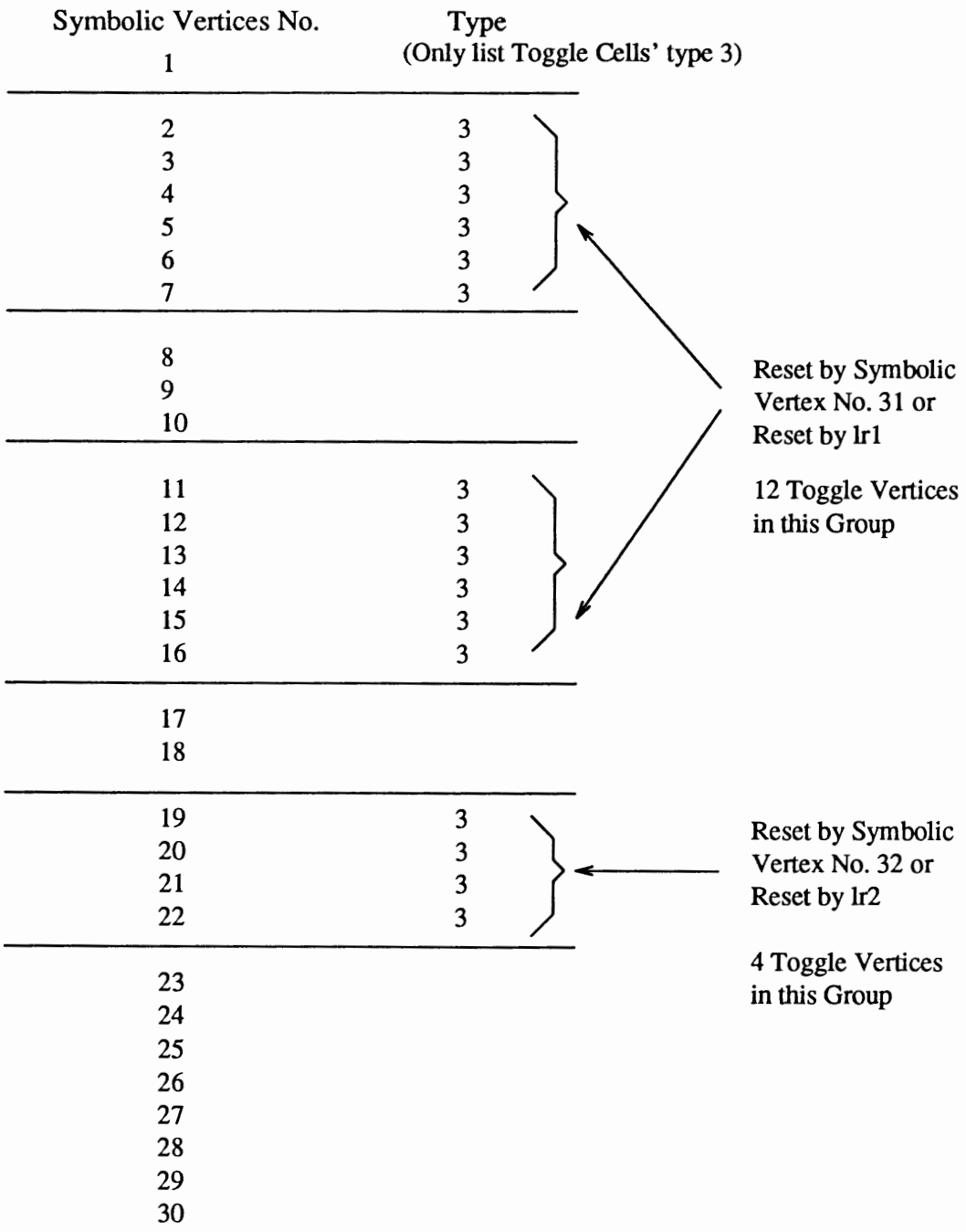


Figure 7. Netlist of "dess2_ba.fit" File.

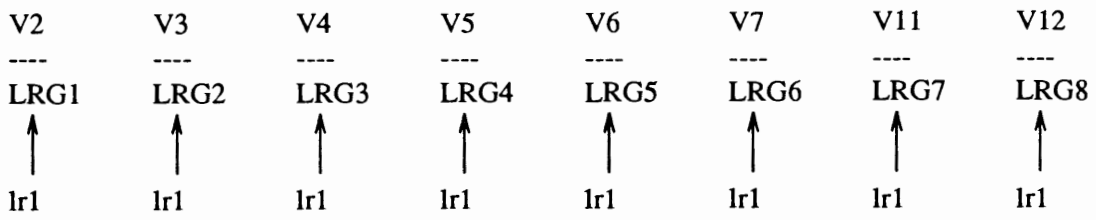


Figure 8. "Split-local-reset-group" Problem.

symbolic local reset lr2. In Figure 8, all the vertices assigned to P_3 partition are from the first toggle sub-group, and they are all triggered by the symbolic local reset lr1. But in the CY7C361 chip, we get only 8 local reset groups(LRGs). Although the symbolic local reset lr1 can be split into several LRGs, it can not dominate all eight LRGs because the symbolic local reset lr2 has to dominate at least one LRG.

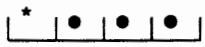
Although we can use a greedy approach such as examining all the vertices in each possible P_3 assignment, it is very time consuming and ineffective.

Therefore, if we can find some outstanding vertices in the toggle cell groups, we can put them at the beginning of the searching queue because these outstanding vertices from the Toggle Group are likely to become the physical global macrocells . Then we put the remaining toggle vertices at the end of the searching queue. Since the remaining toggle vertices are near the end of the searching queue, the poss_ab_rec() function is not likely to choose the set of vertices which can cause the "split-local-reset-group" problem assignment at the beginning of the searching space. Although such a problem will happen (or tend to happen) at the end of the searching space, it is possible that a feasible solution will be found before the program accesses such a "bad" assignment. Therefore, we do not have to add any checking technique in the program to take out such a "bad" assignment. So, the program will not be slowed down.

Based on the analysis of the real feasible solution files found by the PABFIT, we

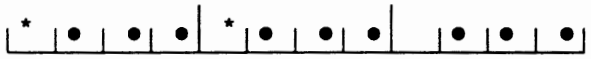
found the toggle vertices from the same toggle sub-group tend to be mapped into the macrocells in the same local reset group or in the adjacent local reset groups of the physical chip. Therefore, when assigning these toggle vertices into the P_3 Assignment, we should take into account the relationship of the vertices in the same toggle sub-group and estimate a good splitting of these vertices among different LRG groups to achieve the best benefit. These estimations are shown in Figure 9.

- toggle sub-group length <= 4



if one toggle vertex is put into P3, it is probable that the other toggle vertices in the same sub-group may not be put into P3;

- 5 <= toggle sub-group length <= 11



if one toggle vertex is put into P3, it is probable that another toggle vertex in the same toggle sub-group may be put into P3;

- toggle sub-group length >= 12

if one toggle vertex is put into P3, it is probable that (toggle sub-group length /4) toggle vertices in the same toggle sub-group may be put into P3 Assignment.

Figure 9. Estimation of the Vertices in the Toggle Group.

IV.1.4 Properties of Chain Group Vertices

Actually, the toggle vertices estimation can be considered as a special case of the vertex ordering. The estimation of the toggle group vertices deals with arranging only

toggle group vertices into the proper location of the vertex searching queue to achieve the goal of speed optimization. When taking into account the outstanding vertices with much higher Total-Degree and Multiple-Degree, we regard the outstanding vertices as the most possible P_3 assignment candidate vertices and regard the vertices coming from the toggle group estimation as the least possible P_3 assignment candidate vertices. In order to achieve the best benefit, we shift the estimated Toggle Group vertices after the outstanding vertices in the vertex searching queue.

For all the vertices in a chain, if one vertex is chosen into P_3 assignment, every fourth vertex after such chosen vertex should be put into P_3 . Vice versa , if one is not chosen into P_3, every fourth vertex after such unchosen vertex can not be put into P_3. Since there are at most 4 vertices in one local reset group(LRG), there exists at least one global vertex in a chain longer than 4 (or a chain with more than 4 vertices). This is shown in Figure 10. We call the chain with more than 4 vertices the long Chain, the chain with only 2 or 3 vertices the short Chain. So if there is a long chain in an input net-list, every reasonable P_3 assignment should contain at least one vertex from this long chain. Therefore, we can put all the long chains at the top of the searching queue and trace all the vertices in each P_3 assignment in order to make sure that each P_3 assignment contains at least one vertex from the long chain. If the P_3 assignment searching pointer has already moved off the long chains range, stop the searching because no reasonable P_3 assignment can lead to a feasible solution without choosing at least one vertex from the long chains.

If there are some outstanding vertices in a long chain, we should consider the so called chain look up order. The chain look up order indicates how the P_3 assignment tries to pick up the vertices from the long chain under the strong C_IN chain connection constraints. There are only 4 combinations of look up orders for any long chains, which is shown in Figure 11. Obviously, as shown in Figure 10, if we try to assign the outstand-

ing vertex into the P_3 assignment at first, we should set the chain look up order as [3,1,2,4] in order to make the algorithm achieve the goal of speed optimization.

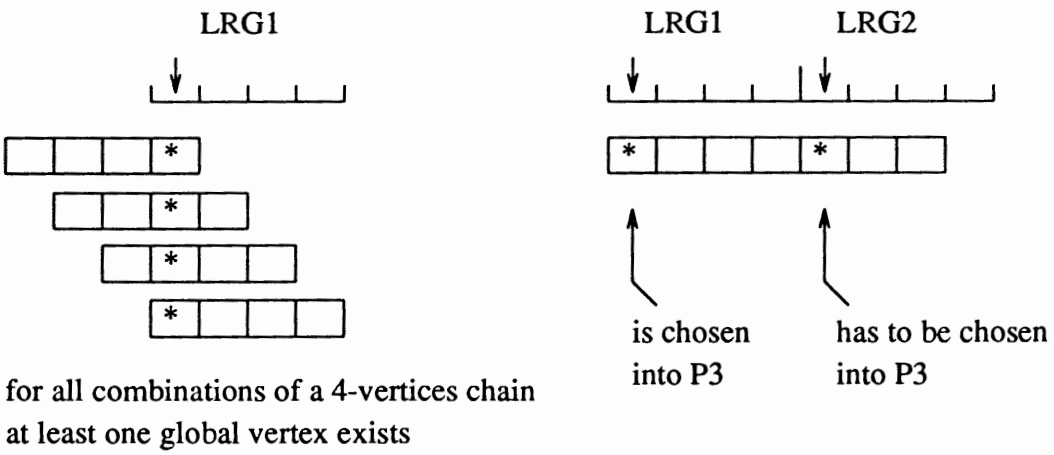


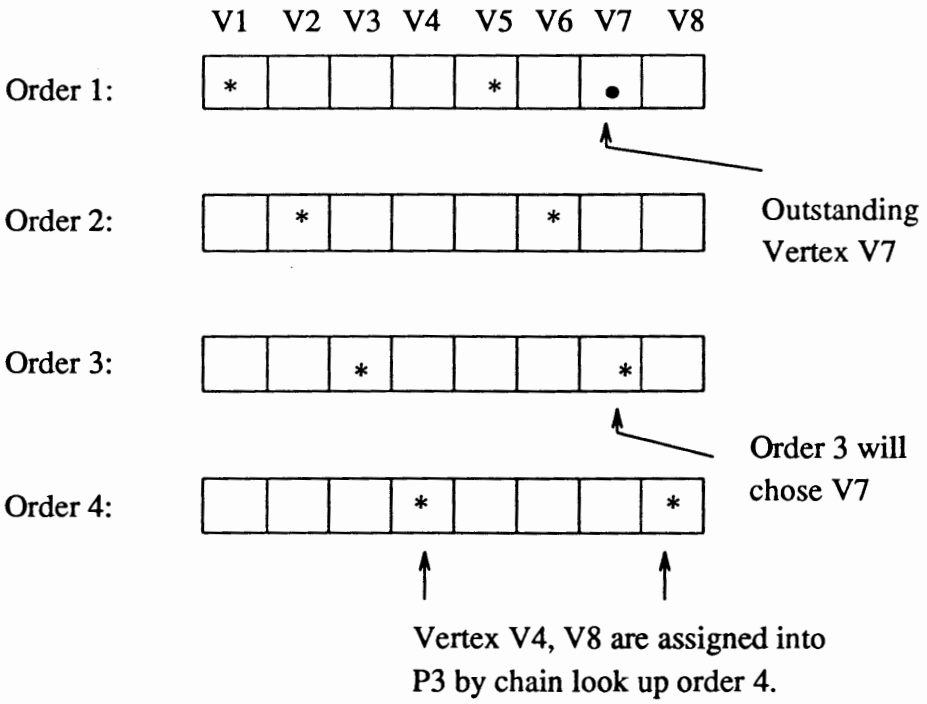
Figure 10. Properties of the Chain Group Vertices.

IV.1.5 Proposed Searching Order

From the above analysis, we know that if there exist long chains, the searching space for the example with vertices in long chains is much smaller than the example with vertices only in the other groups and a reasonable P_3 assignment should contain at least one vertex from the long chain. Therefore, long chains should be put at the top of the vertex searching queue.

The outstanding vertices with much higher Total-Degree and Multiple-Degree are the most possible P_3 assignment vertices candidates, therefore, we can put the outstanding vertices just after the long chains in the vertices searching queue. For the short chains with outstanding vertices, we can just break up the short chains and choose the outstanding vertices.

It is possible that there are some toggle vertices in the outstanding vertices group. In order to achieve the best results, some of the vertices from the same toggle sub-group



natural chain look up order [1,2,3,4]

heuristic chain look up order [3,1,2,4]

Figure 11 Chain Look Up Order.

related to the outstanding toggle vertices may be arranged just after the outstanding vertices group.

In order to avoid the "split-local-reset-group" problem, the remaining Toggle Group vertices may be put at the end of the searching queue. Therefore, the Single Group vertices and the broken short chain vertices may be arranged just before the remaining Toggle Group vertices.

We can draw a conclusion that the best searching order should be:

1. [Long chain vertices]

2. [Outstanding vertices]

3. [Estimated Toggle Group vertices]

4. [Single Group vertices

   & broken short chain vertices]

5. [Remaining Toggle Group vertices]

IV.2 THE NUMBER OF VERTICES ASSIGNED TO P_3

There are 32 state macrocells in a CY7C361 chip and 8 of those 32 macrocells are global macrocells. Therefore, for a certain input netlist file, if the total number of vertices is S ($0<S\leq 32$), then the minimum number of global vertices K must be $K = S-32+8 = S-24$. If $S\leq 24$, then $K=0$. So $0\leq K\leq 8$. The total number of vertices chosen by P_3 assignment is remarked as m.

The original P_3 assignment in PABFIT will first choose $m=K$ global vertices, then increase m step by step until $m=8$. This order can be remarked as [$K \rightarrow 8$]. In the heuristic PABFIT.h algorithm, new heuristic methods are applied in P_3 assignment stage to achieve the goal of speed optimization. So, for a certain input file, if the outstanding vertices number is P , ($P<8$) and if the possible global vertices number is from K to 8, ($0\leq K<P$), we can anticipate that there should be P global vertices in this input netlist file. Instead of allowing the P_3 assignment to do the search by choosing [$K \rightarrow 8$] global vertices, we can force the P_3 assignment to do the search by choosing [$P \rightarrow 8$] global vertices at first, then choosing [$K \rightarrow P$] global vertices. It is likely that a feasible solution will be found by just assigning P global vertices into the P_3 assignment.

For the example files with more than 30 symbolic state vertices, we should consider some other possibilities. It will be explained in example "dess2_ba.fit" input file,

which contains 30 symbolic vertices. The number of the vertices that can be put into P_3 assignment is from 6 to 8. If we put 6 vertices to P_3, the meaning is shown below.

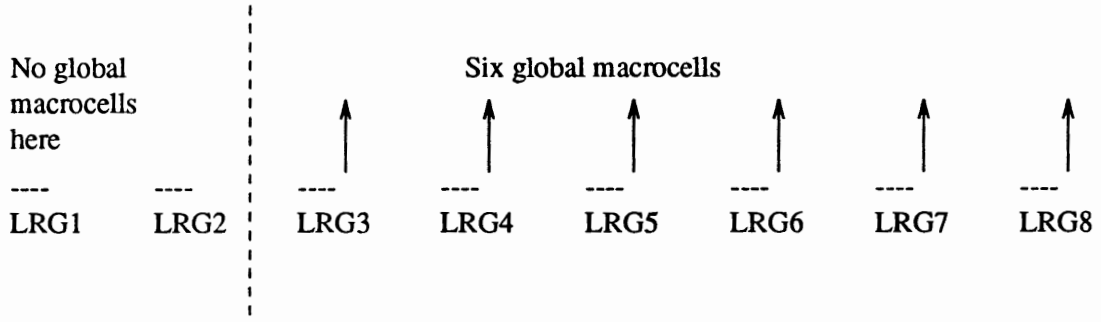


Figure 12 Assignment of Six Global Macrocells

But actually, a global macrocell is more flexible than a local macrocell or intermediate macrocell because it has more output possibilities. So, we can assign 8 vertices to P_3 instead of 6.

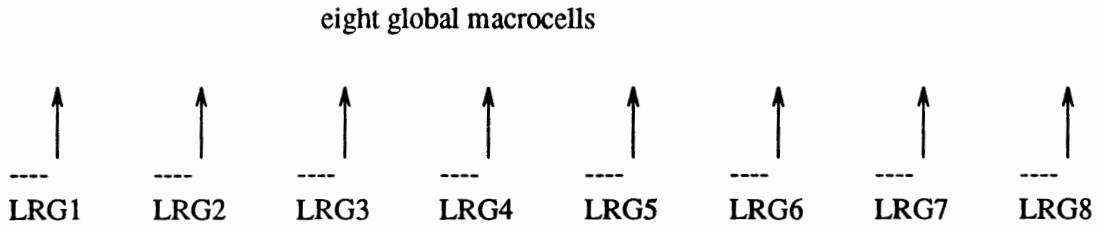


Figure 13. Assignment of Eight Global Macrocells.

By applying this heuristic method to the files with more than 30 state vertices, we make the P_3 assignment do the search by choosing [8->6] global vertices instead of choosing [6->8] global vertices. Since the vertices mapped into the global macrocells will be more flexible than those mapped into the other macrocells, this method can make the algorithm find a feasible solution easier and faster.

CHAPTER V

IMPLEMENTATION OF THE VERTEX ORDERING ALGORITHM

V.1 PABFIT.H WITH VERTEX ORDERING

Based on the analysis of the general properties of the input netlist, we now propose the algorithm with vertex ordering heuristic, which will be described below. Instead of doing the P_3 assignment search randomly, we first examine all the vertices in the netlist file and rearrange the vertex order according to the general properties of the input netlist. The goal of the vertex ordering is to form a proper vertices searching queue for the P_3 assignment to achieve speed optimization. The detail description of the vertex examination and vertex ordering is presented below:

(1) Examine all the vertices in the netlist file and partition them into three different groups, Chain Group, Toggle Group and Single Group, according to the vertices' connection characteristic. The Chain Group may consist of several different chains. Each vertex in the Chain Group belongs to only one of these chains. The Toggle Group may consist of several toggle sub-groups, in each of which all the vertices are triggered by the same symbolic local reset. Any non Toggle Type, non C_IN chain connection vertices belong to the Single Group.

(2) Sort all the vertices in the input file according to their Total-Degree and Multiple-Degree in order to distinguish some outstanding vertices with much higher Total-Degree and Multiple-Degree. Put all these outstanding vertices into a separate

outstanding vertices group, which is independent of the three groups in (1).

(3) Check the Chain Group vertices and divide the chains into Long chains and Short chains. If there exists some outstanding vertices in the short chain, break up the short chains and leave the outstanding C_IN chain connection vertices in the outstanding vertices group, but move the other short chain vertices into the Single Group.

(4) Check the Toggle Group vertices. If there exists some outstanding vertices in the Toggle Group, consider these toggle vertices in the same toggle sub-group as the outstanding toggle vertices, and estimate how many toggle vertices may have a stronger relationship with such outstanding toggle vertices. This estimation is based on the value of the Total-Degree and Multiple-Degree and can be stated as: "higher degree, stronger relation." Those estimated toggle vertices can also be put into a separate estimated toggle vertices group.

(5) Based on step (1) - step (5), form the proper vertices searching queue for P_3 assignment search as:

1. [Long chain vertices]

2. [Outstanding vertices]

3. [Estimated Toggle Group vertices]

4. [Short Chain vertices, Single Group vertices
   or broken up short chain vertices]

5. [Remaining Toggle Group vertices]

   from top down.

(6) Based on the new order, we choose the vertices for P_3 assignment, starting with long chain vertices; followed by the outstanding vertices; estimated Toggle Group Vertices; short chain vertices, Single Group vertices or broken up short chain vertices; and the remaining Toggle Group vertices at the end. This assignment order is based on the principle, first pick up the vertices which have to be in P_3, otherwise no feasible solution exits, then the ones with high assignment probability, next, all the other ordinary vertices. The assignment queue is created as follow:

1. m_l_vertices from the long chain, which have to be in P_3;

2. m_o_vertices from the outstanding vertices group with high probabilities;

3. m_r_vertices from the other 3 groups.

A number of vertices chosen for P_3 assignment is $m = m_l + m_o + m_r$ where $m \geq m_l$.

After the vertex ordering stage, we have to decide on the number of vertices assigned to P_3, which is represented as "m." So "m"= total number of vertices assigned to P_3. Based on the properties of the input netlist, we estimate the best choice for m to start with.

1. The upper bound on m is 8, that is the total number of global macrocells on CY7C361 chip.

2. The lower bound on m is 0 or $S-32+8$, where S is the total number of vertices in the input netlist. If $S \leq 24$, lower bound is 0, otherwise, lower bound is $S-32+8$. The lower bound is set by the netlist limitation.

3. A number of long chain vertices which have to be in P_3, is denoted by m_l.

4. Based on Rule 2 and Rule 3, the proper lower bound m_{proper} is the maximum number of [0, S-32+8, m_l].

Based on the above shown limitations, a starting number of V_{Si} chosen to P_3 Partition is $m = m_{proper}$, then m is increased by one until it reaches 8. If there exists a number of outstanding vertices in the netlist file, we use the symbol "P" to indicate the number of the outstanding vertices. If $m_{proper} < P < 8$, we can use the heuristic method here to assign the $m = P$ number of V_{Si} to P_3 Partition first, then increase m by one until it reaches 8, if the solution is not found, m will be decreased by one form P to m_{proper}.

Therefore, two heuristic strategies are added to the original PABFIT to achieve the speed optimization. The comparison of the flowchart of the original PABFIT and that of the new heuristic PABFIT.h is shown in Figure 14.

V.2 CHOOSING THE VERTEX ORDER

V.2.1 Outstanding Vertices

To order all the vertices in the input netlist based on their Total-Degree and Multiple-Degree and to identify "outstanding vertices" with high Total-Degree and Multiple-Degree, we first set up several data arrays to store the original index number of the vertices in the input netlist and their corresponding Total-Degree and Multiple-Degree as well as other necessary information. This is shown in the following list. The relationship of these arrays is shown in Figure 15.

To distinguish the outstanding vertices, we first sort the array Multiple-Degree[Snodes] and the array Total-Degree[Snodes] from the highest value to the lowest value based on the Multiple-Degree and Total-Degree, respectively. And then re-arrange their corresponding input vertices' natural entry order in the array Multiple-Degree-Order

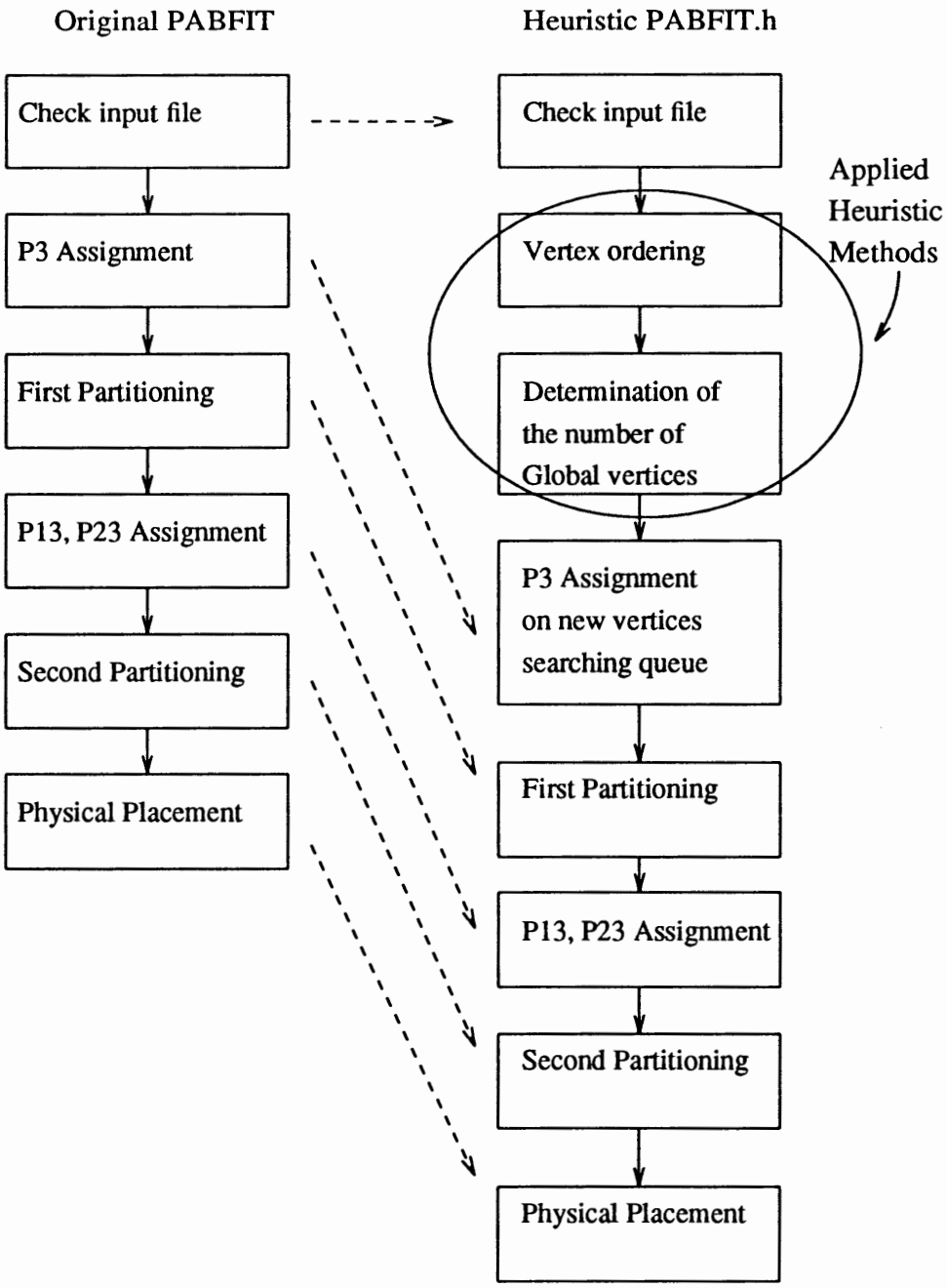
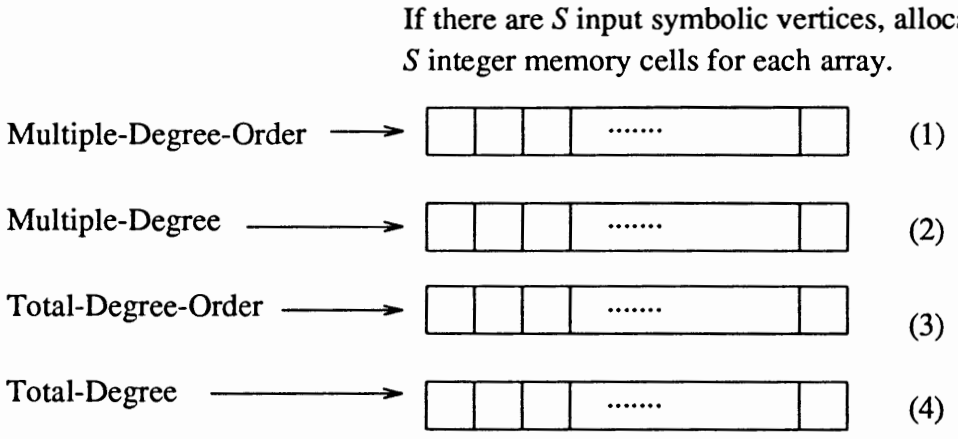


Figure 14. Algorithm Comparison.



(1) store the natural entry order of the input vertices

(2) store the multiple degree of the corresponding input vertices

(3) store the natural entry order of the input vertices

(4) store the total degree of the corresponding input vertices

Snodes:

integer number used for total number of input vertices

Multiple_Degree[Snodes]:

multiple degree of the input vertices

Multiple_Degree_Order[Snodes]:

Natural entry order of the input vertices for the corresponding multiple degree

Total_Degree[Snodes]:

total degree of the input vertices

Total_Degree_Order[Snodes]:

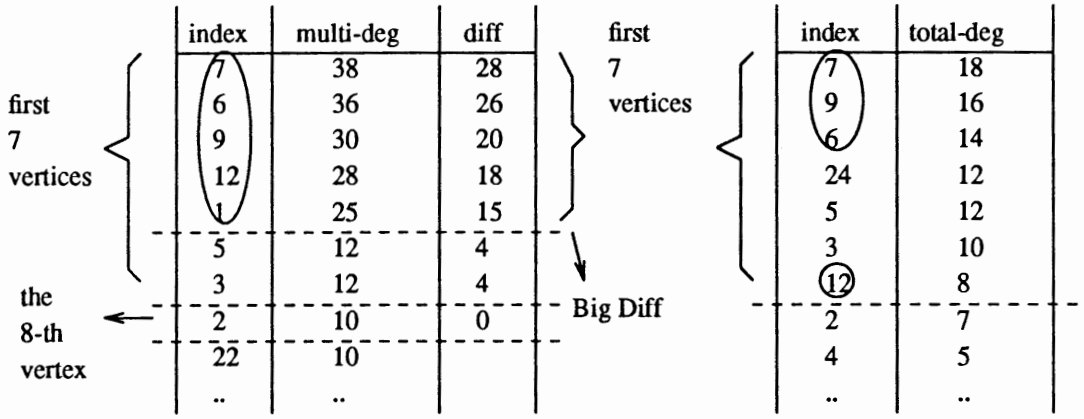
Natural entry order of the input vertices for the corresponding total degree

Figure 15. Arrays for the Outstanding Vertex Search.

[Snodes] and the array Total-Degree-Order[Snodes], according to the vertex degree.

example: input vertices=28; possible global vertices number is 4 to 8; Since there are 4 vertices in one LRG at most, there will be at least 7 LRG occupied. So the number of global vertices should be around 7;

let the rough number of global vertices = 7;



- compare the multiple degree of the first 7 vertices (v7,v6,v9,v12,v1,v5,v3) with the 8-th vertex (V2) in order to find the vertices with big multiple-degree differences. (Big Difference >= 5)
- Find v7,v6,v9,v12,v1 have big differences in multiple degree;
- check the first 7 vertices in Total-Degree[Snodes] in order to check whether v7,v6,v9,v12,v1 exist in the array and find that v7,v9,v6,v12 exist;
- So v7,v9,v6,v12 are regraded as the most possible candidates for global vertices.

Figure 16. Choosing the Outstanding Vertices.

Second, roughly determine the number of global vertices. Since at most there are 4 vertices in one LRG, so there are at least k LRGs, where k = total number of input vertices ÷ 4. Therefore, the global vertices number is probably around k. Since the index order in the array Multiple-Degree-Order[Snodes] and the index order in the array Total-

Degree-Order[Snodes] are arranged according to the vertex degree, the global vertices can be identified from the first k vertices of both arrays because the global vertices should have higher Multiple-Degree and Total-Degree.

Third, compare the multiple degree of the first k vertices in the Multiple-Degree[Snodes] with the $(k+1)-th$ vertex. If there exits a large difference between them, (for instance, difference ≥ 5), mark the vertex with large difference out of these first k vertices in the Multiple-Degree-Order[Snodes] queue.

Last, check the first k vertices in the Total-Degree-Order[Snodes] queue. If the marked vertex in the Multiple-Degree-Order[Snodes] queue does not appear in the Total-Degree-Order[Snodes] queue, clear the mark; therefore, such finally marked vertices are considered outstanding vertices. We use the symbol t to indicate the number of the finally marked vertices.

Figure 16 shows an example how the outstanding vertices are identified.

Here is the pseudo code for distinguishing the outstanding vertices.

```
/**** distinguish the outstanding vertices  *******/

int Snodes; /* total number of input vertices */
int Multiple_Degree[Snodes]:
   /* multiple degree of the input vertices */
int Multiple_Degree_Order[Snodes]:
   /* symbolic index order of the input vertices for the
      corresponding multiple degree */
int Total_Degree[Snodes]:
   /* total degree of the input vertices */
int Total_Degree_Order[Snodes]:
   /* symbolic index order of the input vertices for the
      corresponding total degree */

(1) Sort Multiple-Degree[] from the highest value to the lowest value;

(2) Re-arrange Multiple-Degree-Order[] so that the vertices pointed by
   Multiple-Degree-Order[] has the same decreasing multiple degree
   order as shown by Multiple-Degree[];

(3) Sort Total-Degree[] and re-arrange Total-Degree-Order[] similarly
   to (1) and (2);
```

```
mark_num=0;  /* number of vertices first marked */
check_num=Snodes/4; /* the number of vertices being checked */

for (i=0; i<check_num; i++)
  {
   if ((Multiple_Degree[i]-Multiple_Degree[check_num])>=Big_Difference)
       mark_num++;
  }

t=0;  /* number of vertices finally marked */

for (i=0; i<mark_num; i++)
   {
    for (j=0; j<mark_num; j++)
        {
         if (Total_Degree_Order[j]==Multiple_Degree_Order[i])
           {
           judge the vertex i as a outstanding vertex;
            t++;
           }
        }
   }
```

V.2.2 Toggle Group Vertices

Although the global vertex candidate from the Toggle Group vertices dominate only one section of the vertex searching queue, the process to choose these candidate vertices is more complicated than for other groups of vertices.

First, we identify all the local reset groups in the Symbolic Graph representing the input netlist. If a local reset is found, examine all the toggle vertices triggered by this local reset and store their symbolic index order in a double pointer data structure *A*, which is shown in Figure 17. The data structure *A* is organized as follow, if there are n local resets, the double pointer **toggle-grp points to n different single pointers *toggle-grp[i] ($0 \le i \le n-1$) and each single pointer *toggle-grp[i] points to maximum 32 integer memory cells. If the i-th local reset triggers k_i ($0 \le k_i \le 32$, $0 \le i \le n-1$) toggle vertices, we use the first k_i out of the maximum 32 integer memory cells to store the symbolic index order of the toggle vertices.

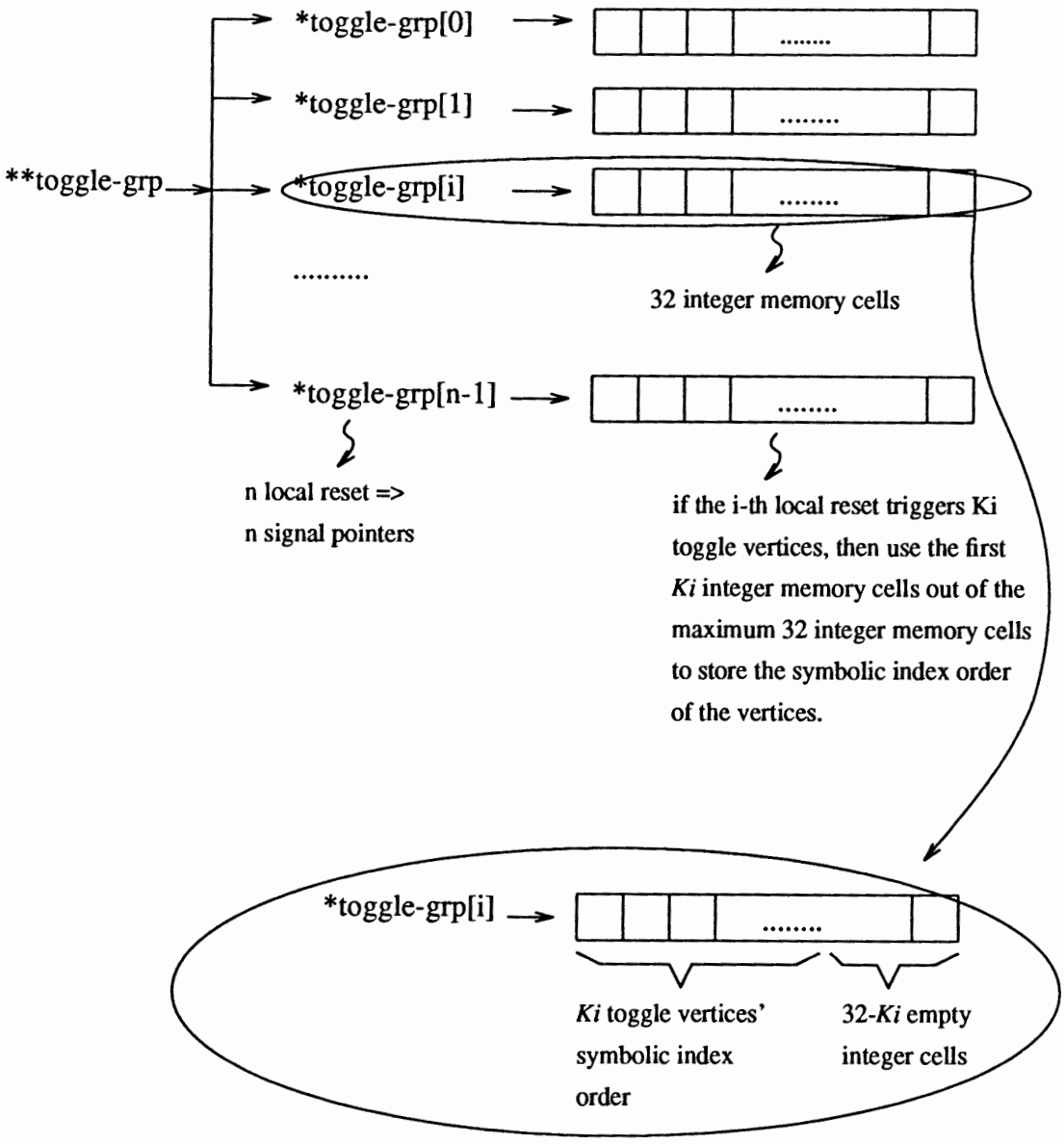


Figure 17. Data Structure A.

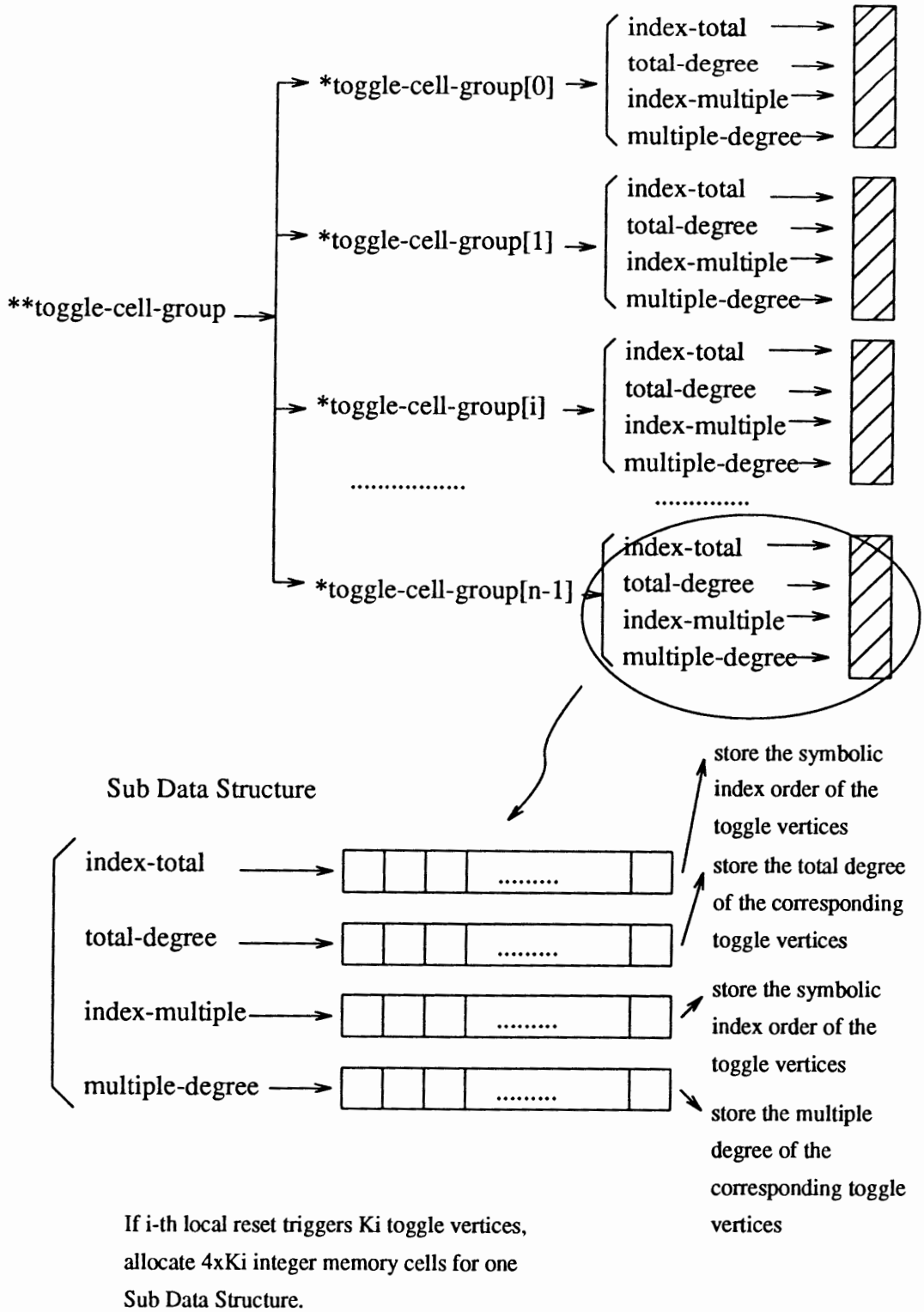


Figure 18. Data Structure B.

Then we set up another double pointer data structure *B*. The double pointer ** toggle-cell-group points to *n* different single pointers *toggle-cell-group[i] ($0 \le i \le n-1$), and each single pointer points to a sub data structure of 4 fixed pointers. These 4 fixed pointers in each sub-data structure *toggle-grp[i] point to four groups of integer memory cells, which store the multiple degree, total degree and their corresponding symbolic index orders for k_i toggle vertices triggered by the local reset i, shown in Figure 18. Actually, each sub-data structure of data structure *B* stores the information of all the vertices in one toggle sub-group of the Toggle Group. In other words, one sub data structure of data structure *B* represents one toggle sub-group.

After having established these two data structures *A* and *B*, we call the procedure "Calculate Total-Degree" and "Calculate Multiple-Degree" to calculate Total Degree and Multiple Degree for each toggle vertex in data structure *A* and transfer the corresponding information into data structure *B*. Sort the data structure *B* from the highest value to lowest value for the total-degree and multiple-degree respectively and re-arrange their corresponding toggle vertex's symbolic index order according to the value of the degree.

Finally, search all the toggle vertices in each sub data structure of data structure *B* in order to mark the outstanding vertices found. Since each sub data structure of data structure *B* represents one toggle sub-group, that is the same as to mark all the outstanding toggle vertices in each toggle sub-group. Some toggle sub-group may contain outstanding vertices, some may not. Then we can choose the candidate global vertices from the toggle group according to the following rules.

(1) If there are at most 3 toggle vertices in one toggle sub group, ignore this toggle sub-group no matter whether outstanding vertex exist in this group.

(2) If there are more than 3 but less than 8 toggle vertices in one toggle sub-group, ignore this toggle sub-group only when there is no outstanding vertex in this sub-group. If there exists one outstanding vertex, search all the toggle vertices in the tog-

gle sub-group except the outstanding vertex and put the one with the highest total-degree and multiple-degree in the estimated Toggle Group vertex.

(3) If there are more than 8 toggle vertices in this toggle sub-group, the number of the estimated toggle vertices should be E, where E is equal to the total number of vertices in the toggle sub-group divided by 4. The process to estimate the Toggle Group vertices is similar to (2).

(4) If there are more than 2 outstanding vertices in one toggle sub-group with less than 8 toggle vertices, we only leave one outstanding vertex in the toggle sub-group, move all the other outstanding vertices with less total-degree or multiple degree to the Single Group.

(5) If there are more than 3 outstanding vertices in one toggle sub-group with more than 8 toggle vertices, only two outstanding vertices are left in the toggle sub-group, move all the other outstanding vertices with less total-degree or multiple degree to the Single Group.

Since the outstanding vertices are placed right after the long chain vertices in the searching queue, they are quite near the top of the vertex searching queue. In order to reduce the probability of "Split-local-reset" problem at the beginning of the P_3 assignment search, we try to maintain some balance and only leave a reasonable number of outstanding vertices in the outstanding vertices group. This rejudgement of the outstanding vertices is combined with the estimation of the toggle vertices and can be regarded as one part of the estimation.

The pseudo code of the estimation of the Toggle Group vertices is listed below.

```
/***** estimation of Toggle Group vertices ********************/

int num_sym_lr; /* the number of symbolic local reset */
int num_gl;  /* the number of symbolic global reset */
int t=0;  /* the index number for toggle sub-group */

for (i=nodes+num_sym_lr+num_gl-1; i>=0; i--)
  { if (Type[i]==4)
      /* Type 4 indicate the local reset */
    {  l=0;
       for (j=0; j<MAX_SIZE; j++)
        { if (Symbolic_Graph[i][j]==1)
        /* Symbolic_Graph is the input netlist */
          { toggle_grp[t][l]=j;
            l++;
          } }
        num_toggle[t]=l;
       /* store the number of toggle vertices in one
          toggle sub-group */
        t++; }
       if (t==num_sym_lr)
        break; }
for (i=0; i<num_sym_lr; i++)
 { for (j=0; j<num_toggle[i]; j++)
   { k=toggle_grp[i][j];
     toggle_cell_group[i]->index_total[j]=k;
     toggle_cell_group[i]->index_multiple[j]=k;
     toggle_cell_group[i]->total_degree[j]=Gb_In_Degree[k]+Gb_Out_Degree[k];
     toggle_cell_group[i]->multiple_degree[j]=
                    (Gb_In_Degree[k]) * (Gb_Out_Degree[k]); }
    /* the In-degree and Out-degree have been stored in
       Gb_In_Degree[] and Gb_Out_Degree[] */ }
initial toggle_mark[]=0
/* indicate one Sub-Toggle Group has not been checked */
for (i=0; i< check_num; i++)
  { if (Original_Order[Total_Degree_Order[i]]==3 &&
       nod[Total_Degree_Order[i]->Type==3 )
       /* indicate the outstanding toggle vertices */
    { for (k=0; k<num_sym_lr; k++)
       {  if (Total_Degree_Order[i]==
             toggle_cell_group[k]->index_multiple[0]) &&
             toggle_mark[k]==0)
       /* find the outstanding toggle vertices in its Sub-Toggle-Group */
          {  toggle_mark[k]=-1;
             if (num_toggle[k]>8)
               { for ( j=1; j<=num_toggle[k]/4-1; j++)
                  { Original_Order[toggle_cell_group[k]->
                    index_multiple[j]]=2;
                    /* estimate the toggle vertices */
                  } } } } } }
```

V.2.3 Vertex Ordering Implementation

After determining the outstanding vertices and examining the toggle vertices, we determine the vertex order. First, we define an array Original-Order[i] ($0 \le i \le S-1$) and initialize it with "0." The index number i of this array indicates the natural entry order of the vertices in the input netlist. Then we define another array Real-Search-Order[i], ($0 \le i \le S-1$). The index number i of this array is a natural number from lower bound to upper bound.

After having set up these two arrays, we examine each vertex i ($0 \le i \le S-1$) and then classify it according to the following rules by marking its corresponding Original-Order[i] with a proper distinguished value w ($1 \le w \le 4$).

Rules:

(1) If the vertex belong to a long chain, marked value $w = 4$;

(2) If the vertex is an outstanding vertex, marked value $w = 3$;

(3) If the vertex is an toggle vertex related to an outstanding toggle vertex, marked value $w = 2$;

(4) If the vertex belongs to a short chain without any outstanding vertices, marked value $w = 1$;

(5) If the vertex is from a broken up short chain, marked value $w = 1$.

We do not have to mark the remaining Toggle Group vertices with $w = 0$ because the array Original-Order[] has already been initialized with value "0." Therefore, the initialization default to set all the remaining toggle vertices with marked value $w = 0$.

According to the result of the classification, we rearrange the original symbolic index order of the input vertices in the array Real-Search-Order[] from the highest marked value $w = 4$ to the lowest marked value $w = 0$. This process is shown in Figure 19.

The pseudo code for Hierarchic Classification and Vertex Ordering is listed below.

```
/* Vertex Ordering */

for (i=0; i<Snodes; i++)
   {  if (vertex i is a finally marked outstanding vertex)
        original_order[i]=3;
      if (vertex i is a estimated toggle vertex related to
         a outstanding vertex)
        original_order[i]=2; }
for (i=0; i<Snodes; i++)
 /* re-judge the long chain with outstanding vertices */
 { if (original_order[i]==3 && nod[i]->chain_length>=4 )
    /* distinguish long chain vertex in outstanding vertex */
    { if (nod[i]->num_in_chain==1)
         for (j=0; j<nod[i]->chain_length; j++)
           original_order[i+j]=4;
       else if (nod[i]->num_in_chain==nod[i]->chain_length)
         for (j=0; j<nod[i]->chain_length; j++)
          original_order[i-j]=4;
      else {
         for (j=0; j<nod[i]->chain_num; j++)
           original_order[i-j]=4;
         for (j=1; j<(nod[i]->chain_length-nod[i]->chain_num+1); j++)
           original_order[i+j]=4;
          }
    }
  /* break up the short chains */
  else if (original_order[1]==3 && nod[i]->chain_length==2)
     { if (nod[i]->num_in_chain==1)
         original_order[i+1]=1;
       else
         original_order[i-1]=1; }
  else if (original_order[i]==3 && nod[i]->chain_length==3)
       {
        if (nod[i]->num_in_chain==1)
        {  original_order[i+1]=1;
           original_order[i+2]=1;  }
        else if (nod[i]->num_in_chain==nod[i]->chain_length)
```

```
        {  original_order[i-1]=1;
           original_order[i+1]=1;  }
        else
        {  original_order[i-1]=1;
           original_order[i-2]=1;  }
       }
  else { if (original_order[i]!=3 && nod[i]->chain_length!=1 )
        {  if (nod[i]->chain_length>3)
             original_order[i]=4;
         /* pick up the long chains */
          else
             original_order[i]=1; }
     }
  if (original_order[i]!=3 && nod[i]->chain_length==1 && nod[i]->type!=3 )
       original_order[i]=1; }
for (i=0; i<Snodes; i++)
  /* vertex ordering in Real_Search_Order[] */
  {
   switch(original_order[i]) {
    case 4:  long_chain_num++;
             break;
    case 3:  outsd_grp_num++;
             break;
    case 2:  estimated_tgl_num++;
             break;
    case 1:  single_grp_num++;
             break;
    case 0:  toggle_grp_num++;
             break;     } }
h=0;
j=h+long_chain_num;
k=j+outsd_grp_num;
l=k+estimated_tgl_num;
m=l+single_grp_num;
for (i=0; i<Snodes; i++)
  { switch(original_order[i]) {
    case 4:  Real_Search_Order[h]=i;
             h++;
             break;
    case 3:  Real_Search_Order[j]=i;
             j++;
             break;
    case 2:  Real_Search_Order[k]=i;
             k++;
             break;
    case 1:  Real_Search_Order[k]=i;
             k++;
             break;
    case 0:  Real_Search_Order[m]=i;
             m++;
             break;     } }
```

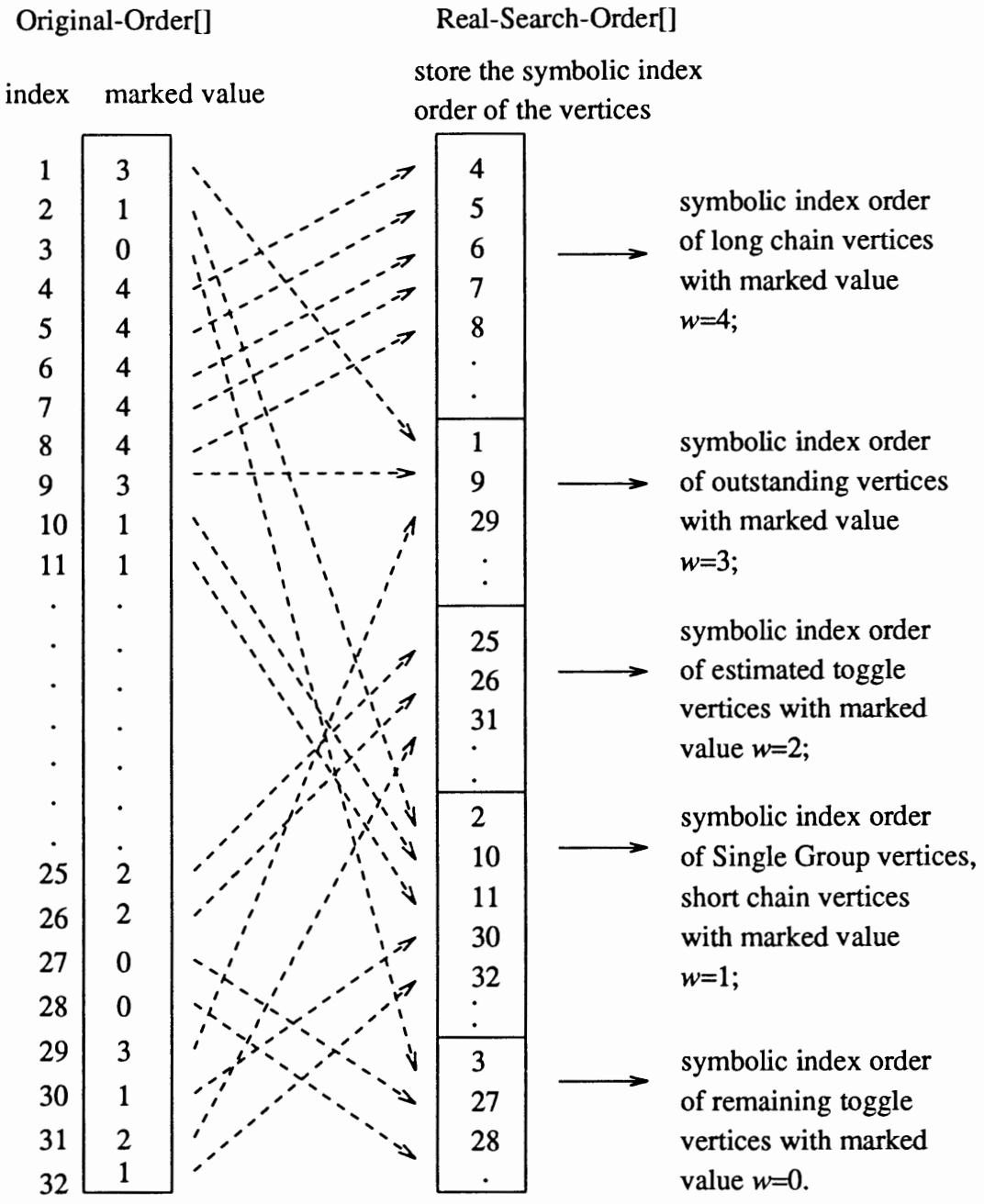


Figure 19. Vertex Ordering.

V.3 CHOOSING THE NUMBER OF GLOBAL VERTICES

The determination of the number of global vertices is much simpler than the vertex ordering. We already know that the lower bound of the possible global vertices is 0 or S-32+8, the upper bound is 8. For example, with long chains, there are at least m_l vertices from the long chains, which have to be in any reasonable P_3 assignment. Therefore, the proper lower bound m_{proper} is the maximum number of [0, S-32+8, m_1]. Usually, we can start with m_{proper} and increase m one by one until it reaches 8. However, if there exist outstanding vertices, to achieve speed optimization, we start with m=P, where P is the number of outstanding vertices, next increase m one by one until it reaches 8. If the solution is not found, m will be decreased one by one from P to m_{proper}. For those examples with more than 30 total input vertices, we can start with m=8 and decrease m one by one until it reaches 6 in order to make good use of the flexibility of the connectivity of global vertices. Besides this, we can also combine the high degree vector heuristic method as a special case in the determination of the number of global vertices.

The pseudo code for this part is shown below.

```
/***** Determination of the number of global vertices *******/

int ab_que[] /* queue of possible number of global vertices */
int start_ab /* the first member of ab_que[] */

if ( number of symbolic nodes >= 30 )
     /* large number input vertices example */
     {  ab_que[] <=  [ 8,7,6 ];
          if (number of outstanding vertices > 0)  search with real search order;
          else                                     search with original order;
          return; }

if ( number of symbolic nodes is ( 24 < Snodes< 30))
     /* Here decide the start_ab value */
        start_ab = Snodes - 24;
else     start_ab = 1;
   if ( number of symbolic nodes <= 24 and
        the input vertices queue has no long chains  )
```

```
        start_ab = 0
    if ( start_ab <the number of long chains in the input vertices queue)
        start_ab = the number of long chains

if ( start_ab < high degree vector number  And
      outstanding vertices number < high degree vector number )
    {   ab_que[] <= [ high degree vector number,
                high degree vector number + 1,
                 ....
                8,
                high degree vector number - 1,
                 ....
                start_ab ]
        if ( outstanding vertices number > 0 )
              search with real search order;
        else     search with original order;
        return }

if ( start_ab  > high degree vector number  And
          start_ab  <  outstanding vertices number )
    {   ab_que[] <= [ outstanding vertices number,
                outstanding vertices number +1,
                ....
                8,
                outstanding vertices number -1,
                start_ab ]
        if ( outstanding vertices number > 0 )
              search with real search order;
        else    search with original order;
        return; }

if  ( start_ab  >  outstanding vertices number )
      {   ab_que[] <= [ start_ab,
                 start_ab + 1,
                 ...
                 8 ]
          if ( outstanding vertices number > 0 )
               search with real search order;
          else   search with original order;
          return; }
```

CHAPTER VI

RESULTS OF THE NEW VERSION OF PABFIT

The PABFIT.h program that was presented in Chapter V has been tested on several industrial examples. These examples are real life test examples obtained from Cypress Semiconductor Inc. The results of PABFIT.h are compared with the results of PABFIT so as to verify the speed benefit achieved by the PABFIT.h program which utilizes the vertex ordering and global vertices number estimation.

Tables II, III, IV, V, and VI show the examples on which the PABFIT.h and PABFIT were tested. Actually, Tables II - VI can be represented as one Table. However, for convenience, we keep these five separated Tables on different comparison. Table I deals with simple low connectivity examples. Table II presents remarkable high connectivity examples. Chain connection examples are grouped in Table III. Table IV presents toggle group examples. No solution examples are grouped in Table V. Tables II, III, and VI use the same format. The column "Time_old" and "Time_new" indicate the CPU executing time generated by the PABFIT and PABFIT.h, respectively. All the test results with both PABFIT.h and PABFIT were obtained from a Gateway 2000 PC with Intel 486 and DOS 5.1 environment. The measure for the CPU time is seconds. If there is an "NC" in "Time_old," it shows that PABFIT could not verify in a reasonable time whether a feasible solution exits or not. However, as it is shown, a new version of the program PABFIT.h was able to find a solution.

A "feasible solution" of the fitting problem is a feasible placement of each element of the netlist and the symbolic local/global resets to the state macrocell and physical

local/global resets of the chip. If there is no feasible solution, it means that the netlist can not be mapped on the CY7C361 chip, or in another word, the VHDL design does not fit into the CY7C361 physical resource. The column "FIT" on Tables II, III, IV, V, and VI show the results of the PABFIT.h and PABFIT. The answer "yes" means that both PABFIT.h and PABFIT were able to find a state/reset-cell feasible placement; "no" means that both PABFIT.h and PABFIT showed that the netlist of the given examples can't be fitted on the device.

Tables II, III, IV, V, and VI also contain information about the characteristics of the input netlist file, or the Symbolic Graph G_s of the corresponding input netlist. This information can be used to evaluate the complexity of the algorithm for a given example. The number of vertices and the number of edges of the symbolic graph G_s are given in the column "vert" and "edge," respectively. Inside the column "vert," the second and third sub-column also gives information about the number of global and local resets included in the netlist. The whole column represents state macrocells/global resets/local resets. Information about the connectivity of G_s can be obtained from the ratio edge/vertices. If the ratio is large, G_s is highly connected and the partitioning properties of G_s restrict the solution space of the fitting problem very effectively. Furthermore, vertex ordering and global vertices number estimation are used very effectively such that the starting search point is very closed to the feasible solution, if one exists. If the ratio is low, the solution space is only weakly restricted by the connectivity. In this situation, if the netlist does not contain long chains, the solution space of the algorithm may be large. However, since the complexity of the input netlist is considerable low, it should be easy to find a solution to the fitting problem in this case. The chain constraints, especially the long chain constraints can restrict the solution space of the fitting problem very effectively. They can influence the partitioning of the Symbolic Graph G_s very strongly, because chain vertices have to be mapped to adjacent macrocells on the chip and at least

one vertex from a long chain should be presented in the global vertices assignment. Therefore, vertex ordering is applied to shift the long chain vertices to the top of the vertex searching queue for the global vertices assignment stage. Some restrictions can also be added into the PABFIT.h in order to make sure each global vertices assignment contains at least one vertex from the long chain. This will highly restrict the solution space and lead to a solution very quickly. The number of vertices of G_s that have an input chain form other vertices is given in the column "C_IN" in Table II, V, and VI.

TABLE II

SIMPLE EXAMPLES WHERE PABFIT.H AND PABFIT ARE EQUIVALENT

FILE NAME	vert	edge	max conn	C_IN	Time_Old (s)	Time_New (s)	Fit
busa_ba	16/0/0	10	3	8	0.054	0.109	yes
busa_ba1	16/0/0	12	3	10	0.16	0.16	yes
cntr_ba	9/0/1	45	9	0	0.32	0.109	yes
counter1	13/0/0	12	2	1	0.00	0.054	yes
cadman_b	25/0/1	45	11	0	62.02	62.91	yes
cadman_ba	25/0/1	45	11	0	72.14	72.08	yes
demo2	9/0/0	9	3	4	0.054	0.054	yes
epee11	24/0/0	35	8	13	183.62	183.40	yes
epeecon_ba	16/1/0	34	6	8	17.03	18.18	yes
micro_ba	9/1/1	20	9	0	0.054	0.00	yes
examp33	14/0/2	47	9	0	0.43	0.65	yes
mlt_fsm1	21/0/0	22	3	5	0.109	0.109	yes
reword	15/1/1	23	4	5	0.109	0.054	yes
stepper	16/0/0	19	6	8	3.40	3.57	yes
tsr2_bug	18/0/0	19	2	2	0.109	0.109	yes

Table II shows the examples with relative low edge/vertices ratio. Here, both PABFIT.h and PABFIT demonstrate that all the examples in Table II can be fitted into the CY7C361 chip. Comparing the CPU executing time of PABFIT.h and that of PABFIT, we found that PABFIT.h can not be much faster than PABFIT. The executing time of both programs are quite close to each other. Actually, this situation is reasonable

because the solution space for the examples with low connectivity is small and the vertices with particular properties, which are regarded as the most possible global vertices assignment candidates, are not obvious among all vertices form the input netlist file. Besides, since PABFIT.h has to spend some CPU time to form the proper vertex order and choose the proper starting global vertices number, the executing time of PABFIT.h may be a little bit longer than that of PABFIT.

From the results in Table III, we found that the PABFIT.h can find a solution faster than the PABFIT. Especially for the example "dees_ba.fit" and "e019_ba.fit," the improvement is great.

In the example "dees_ba.fit," there are 30 state vertices, including 2 TOGGLE Sub-Groups. One with 12 vertices, another with 4 vertices. The PABFIT.h found 1 outstanding vertex in the 12-vertex TOGGLE Sub-Group, and it estimated that there should be another vertex in this TOGGLE Sub-Group, which is related to the found outstanding vertex and should be shifted to the proper section of the vertices searching queue. So PABFIT.h recognized the vertex with the second maximum Multiple and Total degree as the estimated Toggle vertex. This is described in Chapter.IV and Chapter.V. PABFIT.h also found another outstanding vertex in the TOGGLE Sub-Group with 4 vertices. According to the rules we described in Chapter.III, it ignores the other toggle vertices. The other outstanding vertices are found in the single vertices group. By examining the result of "dees_ba.fit," we found six global vertices, which were used in the solution, among the first 10 location in the Real-Search-Order queue, which is formed by the vertex ordering as the new vertices searching queue. The other 2 real global vertices are in the middle of the Real-Search-Order queue. Since there are 30 state vertices, PABFIT.h chose 8 as the starting global vertices number, that is, the same number of the real global vertices as the Real-Search-Order queue shows. Therefore, it took PABFIT.h only 57.85 seconds to find a feasible solution. This result is much better than that from PABFIT

since "dees_ba.fit" is listed as NP (Not completed) examples for PABFIT.

In the example "e019_ba.fit," there are 30 state vertices and 7 short chains, the PABFIT.h found 3 outstanding vertices in three 2-vertex short chains, respectively. By applying the chain-break-up technique, the vertex ordering procedure leaves the found outstanding chain vertices at the top of the vertex searching queue, but moves the other vertices from the broken up chain to the bottom. The other outstanding vertices were found in a single vertices group. After examining the Real-Search-Order queue, we found that the 8 vertices chosen to a found solution as global vertices are exactly at the first 8 locations of the Real-Search-Order queue. Therefore, PABFIT.h chose 8 as the starting global vertices number and just picked up the first 8 vertices in the new vertex searching queue as the real global vertices. It took PABFIT.h 98.02 seconds to figure out a feasible solution and it is much better than what PABFIT did, which took 5 hours.

Besides these two examples, all the other examples in Table III show that PABFIT.h is faster than PABFIT in the CPU executing time for these type of the netlist files.

TABLE III

EXAMPLES WHERE PABFTI.H IS BETTER THAN PABFIT

FILE NAME	vert	edge	max conn	C_IN	Time_Old (s)	Time_New (s)	Fit
tgen_ba1	25/1/1	60	14	0	3.46	2.52	yes
warpt_ba	32/1/0	64	5	2	28.90	15.38	yes
e019_ba	32/0/2	85	13	8	11176.81	98.02	yes
dees2_ba1	30/1/2	100	13	15	NP	57.85	yes

Table IV shows the fitting results of the C_IN chain examples where PABFIT.h is faster than PABFIT. We can see that Table IV is a little bit different from Table II and Table III. Instead of giving the total number of how many C_IN chain vertices in one test

example, Table IV lists the numbers of how many vertices in each of the sub-chain groups in the test example. Here we used "Chain_num" to indicate this column. For example, "4/5/6" means that there are 3 different chain groups and the first chain contains 3 vertices, the second 5 vertices and the third 6 vertices. For all the examples containing C_IN chains, whether they can be fitted into the device or not, PABFIT.h can find a feasible solution much faster than PABFIT. This proves the correctness of our Heuristic PABFIT.h algorithm. The original PABFIT only calculate how many vertices in a C_IN chain should be put into P_3 Assignment and it chooses those vertices according to their natural entry order. But our vertex ordering PABFIT.h, first estimates which of the vertices in the whole chain is the most possible P_3 assignment candidate or the so-called outstanding vertex. Then it reforms the chain look up order in order to pick up the outstanding C_IN chain vertices and its relative C_IN chain vertices first. Here, the distance between the relative C_IN chain vertices and the outstanding C_IN chain vertex must be 4 or the multiple of 4. Otherwise, they are not related to the outstanding vertices anymore and must be excluded from the P_3 assignment.

TABLE IV

C_IN CHAIN EXAMPLES WHERE PABFIT.H IS BETTER THAN PABFIT

FILE NAME	vert	edge	max conn	Chain_num	Time_Old (s)	Time_New (s)	Fit
chain_cad	25/0/1	45	9	4/	340.3	330.0	yes
chain_d	30/1/2	100	7	2/5/2/3	34.6	30.0	yes
chain_t2	16/1/0	33	4	2/2/4/2/3	21.7	11.6	yes
chain_tr	18/0/0	44	6	2/2/5	51.7	29.6	yes
chain_w2	19/1/1	37	4	3/2/2/7	11.2	8.3	yes
chain_g	25/1/1	60	7	17/	18.2	14.8	no
chain_t	16/1/0	33	4	2/2/2/2/6	93.1	8.1	no
chain_w	19/1/1	43	5	3/2/5/2/2	67.1	36.5	no

TABLE V

TOGGLE GROUP EXAMPLES WHERE PABFIT.H IS BETTER THAN PABFIT

FILE NAME	vert	edge	max conn	C_IN	tgl_num	Time_Old (s)	Time_New (s)	Fit
tgl_e	14/0/2	60	7	0	3/3	70.0	57.5	yes
tgl_kk	20/1/3	57	6	0	4/3/3	612.9	11.0	yes
tgl_ne	25/0/2	68	5	0	7/8	57.3	31.5	yes
tgl_ll	25/0/3	53	5	10	4/3/1	36.3	5.7	yes
tgl_nw	28/0/2	77	5	12	7/4	468.0	343.5	yes
tgl_2	30/1/2	99	7	9	12/4	146.8	39.3	yes
tgl_eb	30/0/2	89	5	15	11/3	7529.7	183.2	yes

Table V shows the fitting results of the toggle group examples where PABFIT.h is faster than PABFIT. Instead of presenting the number of C_IN chain vertices and the number of C_IN chain groups, Table V presents the numbers of toggle vertices in each of the sub-toggle groups. Here we use "tgl_num" to represent this column. Similar to Table IV, the item "6/4" means that there exist two different toggle sub-groups and the first toggle sub-group contains 6 vertices and the second 4 vertices. For all these toggle group examples, PABFIT.h can find a feasible solution much faster than PABFIT. PABFIT does not care whether a chosen vertex is in the toggle group or not, it just pick it up into the P_3 assignment according to the natural entry order. Therefore, a random toggle vertex may be put into the P_3 assignment separately without considering the connection constraints of all the other toggle vertices in the same toggle group. This confuses the fitter and causes the fitter to spend a long time checking the connection constraints. PABFIT.h takes the whole group of toggle vertices into account. It chooses some outstanding vertices from the same toggle group into the P_3 assignment and moves all the other unchosen toggle vertices from the same toggle group to the end of the searching queue. The symbolic connection constraints of the chosen vertices can easily match the physical connection constraints of the state macrocells and local/global reset cells. There-

fore, a feasible solution can be found by PABFIT.h much faster than PABFIT.

Table IV shows the example that both PABFIT.h and PABFIT prove no feasible solution exists. Such designs can not be fitted into the CY7C361 physical device. Therefore, both PABFIT.h and PABFIT have to examine all the possible cases one by one without exception to prove that there is no feasible solution. The executing times obtained from both programs for these examples are almost identical. Since "Data_8.fit" and "Examp11.fit" are simple files with small solution space, the executing times obtained from both PABFIT.h and PABFIT for these two examples are almost the same and they are also very short. "mbarn_ba.fit" contains three long chains, the first with 5 vertices, the second and the third all with 7 vertices, respectively. It also contains 4 2-vertex short chains. Since at least one vertex from the long chain should be placed the global vertices assignment, the starting global vertices number is 3. Although PABFIT.h kept the long chain at the top of the vertices searching queue and used some additional constraints to assure that there are at least three vertex from the long chains respectively in every global vertices assignment, it did not gain much in speed because there are too many long chain vertices. As the possible global vertices number increased from 3 to 8, PABFIT.h did not exclude many assignment cases even though it utilized the long chain constraints, but unfortunately it had to spend time checking the possible assignment. "epee1_res.fit" is an example with high connectivity and contains only 2-vertex short chains. So both PABFIT.h and PABFIT have to go step by step to examine each possible assignment case. Therefore, for the examples with no feasible solution, the executing time is very difficult to cut down.

The PABFIT algorithm does not rely on the properties of the input files. It explores all possible cases of the searching space in the random way. Therefore, for the simple file with small searching space, it will generate the result sooner or later, but for the complex or large searching space files, it will keep on running for long, long time

without producing a feasible answer.

The PABFIT.h algorithm judges the properties of the input files at first. It forms the proper vertices searching queue and chooses the proper starting global vertices number before the real searching. Therefore, it is possible for this new algorithm to generate the feasible result faster than the original algorithm.

TABLE VI

NO FEASIBLE SOLUTION EXAMPLES

FILE NAME	vert	edge	max conn	C_IN	Time_Old (s)	Time_New (s)	Fit
data8	10/1/4	78	15	0	0.98	1.31	no
examp11	13/1/2	70	9	8	0.109	0.16	no
mbarn_ba	31/0/0	28	4	22	129.94	130.05	no
epee1_res	27/0/0	53	13	13	5656.31	5728.35	no

CHAPTER VII

COMPLEXITY ANALYSIS

As we have presented in Chapter III, the new heuristic methods are based on choosing the proper order of how many vertices (number given by m) out of all the input vertices are assigned a to P_3 assignment and are also based on choosing the proper order of how these m different vertices are assigned to P_3. Let n be the number of total vertices and let m be the number of vertices assigned to P_3 assignment. Here, $0 \le m \le n$. Therefore, the number of possible P_3 assignments can be calculated according to the following equation.

$$Number\ of\ possible\ \ P_3\ \ assignments = \sum_{m=0}^{n} \binom{n}{m}$$

If the $i-th$ P_3 assignment tries to assign $m = k_i$ vertices into P_3, here $0 \le m = k_i < n$, there are $\binom{n}{k_i}$ different solution methods to assign these k_i vertices into the P_3 assignment.

Assume that there exists L long chains in the input netlist file and each long chain contains l_i C_IN chain connection vertices, here $L \ge 1$, $4 \le l_i < n$, $\sum_{i=0}^{L} = L_m \le n$. In the worst case, there must be at least one vertex from each long chain in the P_3 assignment. Therefore, the number of possible solutions for the $i-th$ P_3 assignment on the long chain example can be calculated by the following equation. If denote Np_i as the number of

possible solutions for the $i-th\ P_3$ assignment, then,

$$Np_{i\_long\ chains} = \begin{bmatrix} n - L_m \\ k_i - L \end{bmatrix} \prod_{i=1}^{L} \begin{bmatrix} l_i \\ 1 \end{bmatrix}$$

$$Np_{i\_long\ chains} = \frac{l_1\, l_2\, \cdots\, l_L\, (n - L_m)!}{(k_i - L)!(n - L_m - K_i + L)!}$$

$$Np_{i\_long\ chains} < \frac{n!}{k_i!\,(n-k_i)!} = \begin{bmatrix} n \\ k_i \end{bmatrix}$$

So the solution space of the long chain example is much smaller than that of the regular example.

By using heuristic, we assign the outstanding vertices in P_3 assignment first. Denote the number of outstanding vertices as P, $0 \le P \le k_i$. The number of possible $i-th$ P_3 assignment on the outstanding vertices case can be calculated by the following equation.

$$Np_{i\_outstanding\ vertices} = \begin{bmatrix} n-P \\ k_i-P \end{bmatrix} + \begin{bmatrix} n \\ k_i \end{bmatrix} - \begin{bmatrix} n-P \\ k_i-P \end{bmatrix}$$

Obviously, $\begin{bmatrix} n-P \\ k_i-P \end{bmatrix} < \begin{bmatrix} n \\ k_i \end{bmatrix}$. If the PABFIT.h algorithm can find a feasible solution within the first $\begin{bmatrix} n-P \\ k_i-P \end{bmatrix}$ assignments, it will not go on checking the remaining possible assignments. This will achieve the performance speed.

For the Toggle Group vertices, we moved the remaining Toggle Group vertices at

the end of the searching queue after the estimation of the Toggle Group vertices. Assume that there are T Toggle Group vertices. In the worst case, none of them can be assigned to P_3 assignment. Therefore, the number of possible solutions for the $i-th$ P_3 assignment can be calculated by the following equation.

$$Np_{i\_Toggle\ Group} = \begin{bmatrix} n \\ k_i \end{bmatrix} - \begin{bmatrix} n-T \\ k_i-T \end{bmatrix} + \begin{bmatrix} n-T \\ k_i-T \end{bmatrix}$$

Since there are no Toggle Group Vertices in the first $\begin{bmatrix} n \\ k_i \end{bmatrix} - \begin{bmatrix} n-T \\ k_i-T \end{bmatrix}$ assignments, the possibility to encounter the "Split-local-reset" problem within these assignments is zero. Therefore, in this case, we search the non-Toggle type vertices first for a proper P_3 assignment to get the feasible solution.

Obviously, if there is no long chains, whether we use heuristic or not, the number of the possible P_3 assignment and the number of the different solution methods on certain P_3 assignments will not be changed. The whole heuristic algorithm only changes the order of vertices assignment.

If we use a circle to represent the whole searching space, the dots inside the circle represent the possible solution. The PABFIT algorithm is trying to search the whole searching space step by step without any exception. But the heuristic PABFIT tries to guess which section of the whole searching space is the section where the possible solution exists. Then searches this section first. See Figure 20. From the previous theory description and result analysis, we can prove that the heuristic method can speed up the whole PABFIT algorithm. That means, a possible solution can be found by PABFIT.h faster than PABFIT.

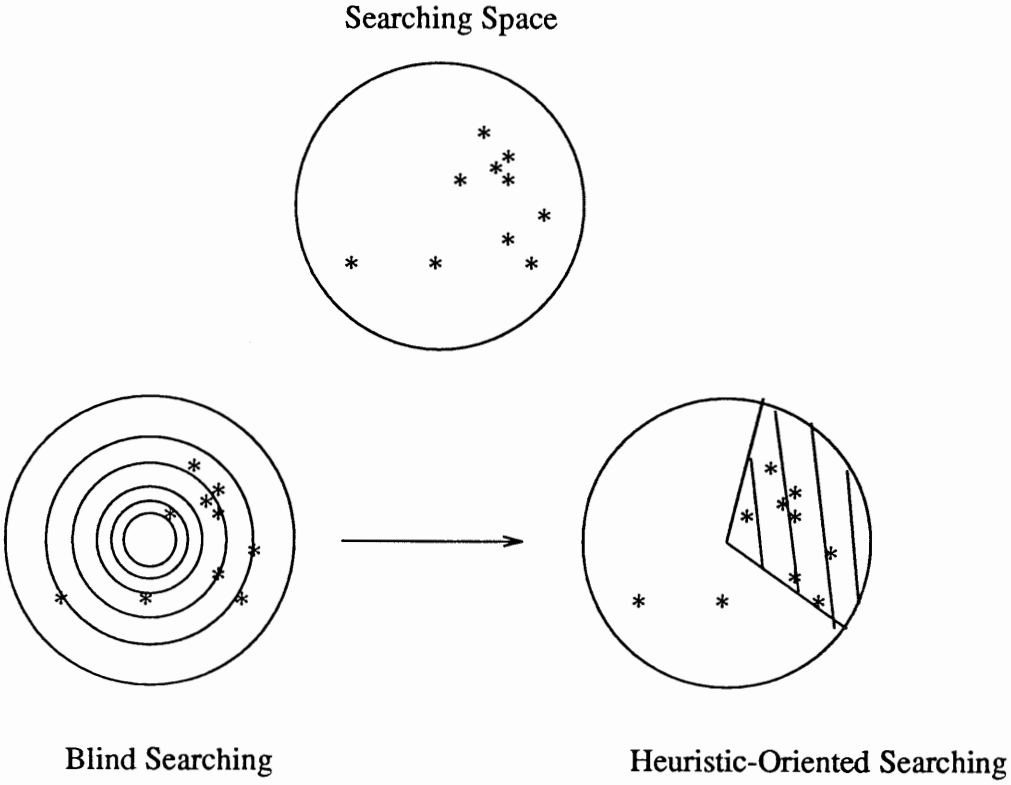


Figure 20. Searching Space.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

In this thesis, we developed a vertex search approach to improve the time complexity of the PABFIT algorithm. These new heuristic methods are developed based on the analysis of the general properties of the chip architecture.

We have introduced a heuristic approach to influence the search order and the algorithm still remains exact.

The test files for the PABFTI.h are the real industry design examples from Cypress Semiconductor Inc. The test results of PABFIT.h shows this program can find a feasible solution much faster than the original PABFIT, especially for the example with high connectivity and containing long chains.

These Architecture-Driven fitting approach and Architecture-Driven speed optimization heuristic methods with some modification can also be applied to the layout problem of the other CPLD devices (Complex PLD) with highly restricted connections.

The PABFIT.h can be separated into two-level partitioning stages, the global vertices assignment & the first-level partitioning, and the intermediate vertices assignment & the second-level partitioning. The un-related assignment combinations of the global vertices assignment can be processed simultaneously by parallel computing. So do the assignment combinations of the intermediate vertices assignment. Therefore, the future work can be concentrated on utilizing the parallel computing technique on this program.

CHAPTER IX

APPLICATION

In this Thesis, we developed an architecture-based partitioning fitting algorithm with vertex ordering heuristic targeted on an EPLD device, the CY7C361 EPLD. The design ideas of this algorithm can also be applied on the layout problem of the other EPLD or CPLD devices with highly restricted segmentation connection architecture, similar to CY7C361 EPLD.

The MAX5000 192-macrocell EPLD, provided by Altera Corporation, is a highly restricted segmentation connection device. The 192 macrocells in MAX5000 are divided into 12 Logic Array Blocks (LABs), 16 pre LAB. There are 384 expander product terms, 32 per LAB, to be used and shared by the macrocells within each LAB. Each LAB is interconnected with a Programmable Interconnect Array (PIA), allowing all signals to be routed throughout the chip.

Each LAB contains 16 macrocells. In LABs A, F, G and L, 8 macrocells are connected to I/O pins and 8 are buried, while in LABs B, C, D, E, H, I, J and K, 4 macrocells are connected to I/O pins and 12 are buried. The buried macrocells are connected to PIA. The architecture of MAX5000 device is shown in Figure 21.

From Figure 21, we can know that MAX5000 EPLD has the partitioning properties in its chip architecture. Therefore, the arrangement of the partitions of the 192 macrocells can be used as an orientation in designing the MAX5000 EPLD layout algorithm. We can divide the vertices in the netlist into different sets according to the partitioning properties of the MAX5000 device and map such sets of vertices into the groups of macrocells in

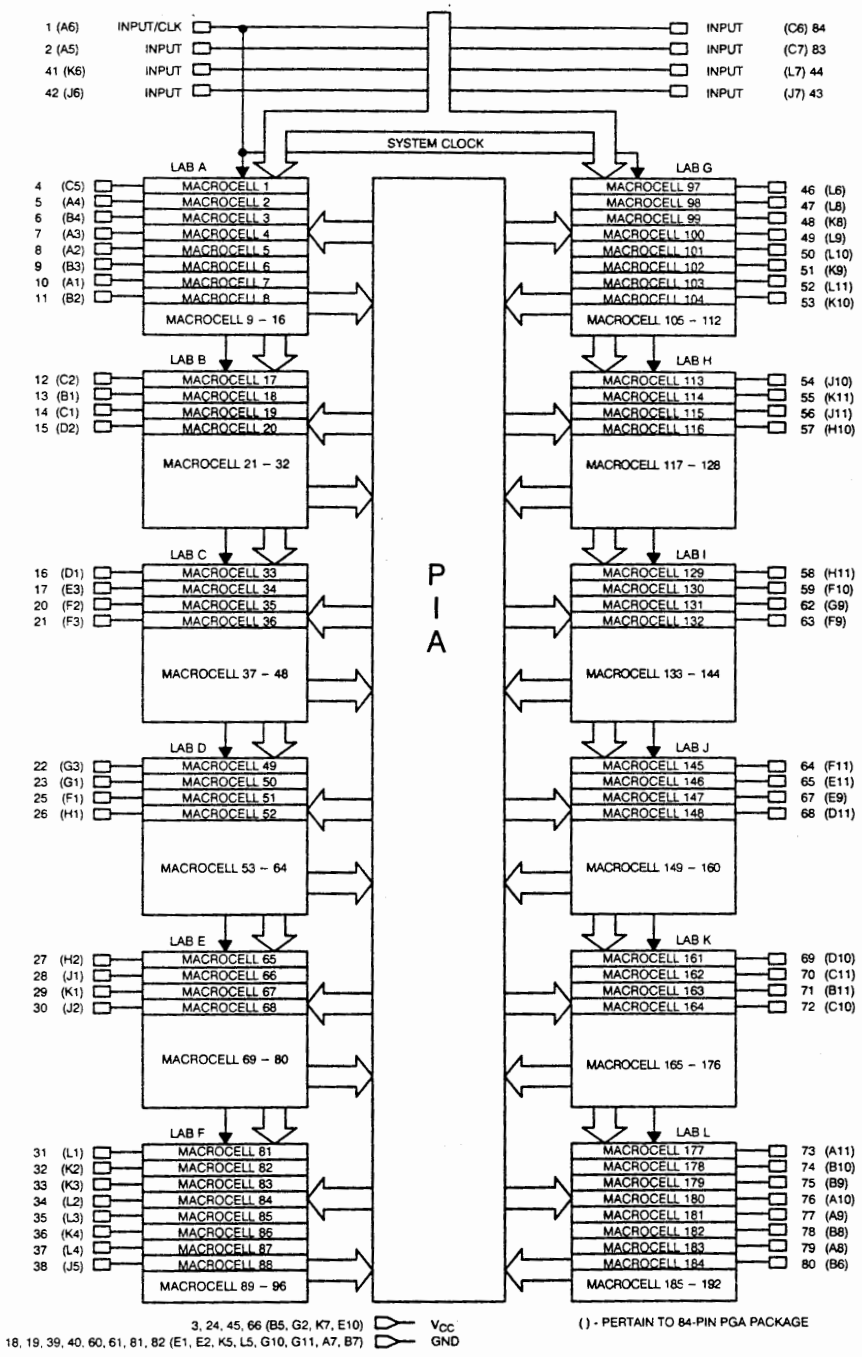


Figure 21. Architecture of MAX5000 EPLD.

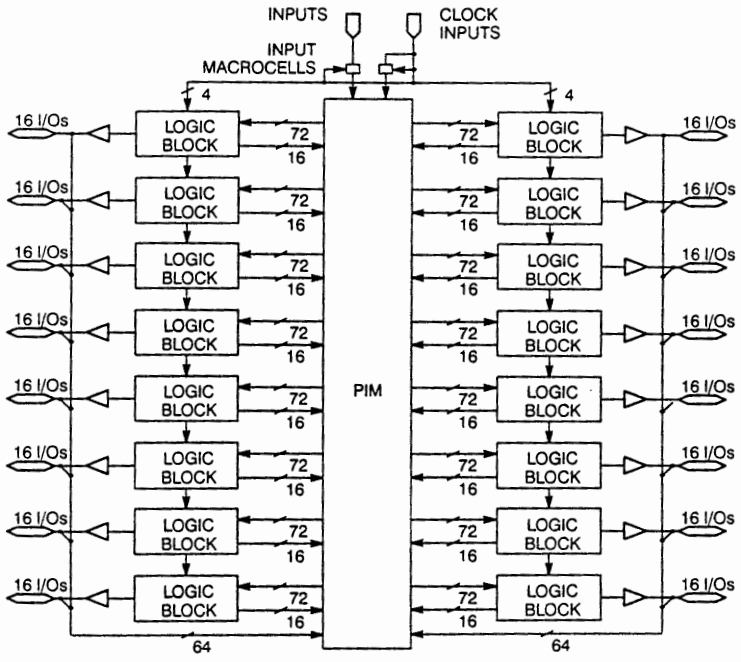


Figure 22. Architecture of CY7C376 EPLD.

proper LABs. Obviously, the LABs with more buried macrocells should have more connections to PIA than the LABs with less buried macrocells. So the ordering of the macrocells with more connections to PIA should be an important factor in the layout problem. Therefore, the design ideas for CY7C361 fitting algorithm can be partially applied or be applied with some modification on the MAX5000 layout problem.

The CY7C376 256-macrocell Flash PlD, provided by Cypress Semiconductor, is an EPLD device with a similar Architecture to Altera's MAX5000 EPLD. The 256 macrocells in the CY7C376 are divided between sixteen logic blocks. Each logic block includes 16 macrocells, a 72 x 86 product term array, and an intelligent product term allocator. The logic blocks in the CY7C376 architecture are connected with an fast and predictable routing resource: the Programmable Interconnect Matrix (PIM). Unlike

MAX5000, each logic block in CY7C376 has only one buried macrocell along with each I/O macrocell. In other word, in each logic block, there are eight macrocells that are connected to I/O cells, eight macrocells that are internally fed back to PIM. Architecture of CY7C376 is shown is Figure 22.

Obviously, from Figure 22, the CY7C376 EPLD also has the partitioning properties in its design architecture. Therefore, its layout problem can use the partitioning fitting concepts, similar to the CY7C361 fitting problem. However, since the logic blocks in CY7C376 are identical to one another, the ordering of such logic blocks are the ordering of macrocells mapped into such logic blocks may not be an important factor on its layout problem.

Conclusionly, the design ideas of the vertex ordering PABFIT.h algorithm can be applied on the layout problem of the other EPLD or CPLD devices with similar architecture as CY7C361 EPLD.

REFERENCES

[1] Anderson, R.E., Coppola, A., Perkowski, M. A., "VHDL Synthesis of Concurrent state machines to a programmable Logic Device", *VHDL International User's Forum, Scottsdale, Arizona*, May 3-6, 1992.

[2] U. Schlichtmann, F.Brglez, M.Hermann, "Characterization of Boolem Function for Rapid Matching in FPGA Technology approaching", *Proc. of ACM/IEEE DAC*, pp. 374-379, 1992.

[3] Murgai, R., Shenoy, N., Brayton, R.K., Sangiovanni-Vincentelli, A., "Improved Logic Synthesis Algorithms for Table Look Up Architectures", *ICCAD-91, pp. 564-567, Santa Clara*, CA, November 1991.

[4] D. Filo, J.C. Yang, F. Mailhot, G.D. Micheli, "Technology Mapping for a Two-Output RAM-based Field-Programmable Gate Array" *European Design Automation Conf.* pp. 534-538, February 1991.

[5] Robert J. Francis, Jonathan Rose, Zvonko Vranesic, "Chortle: Fast Technology Mapping for Lookup Table-Based FPGAs", *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 227-233, San Francisco, CA, June 1991.

[6] Robert J. Francis, Jonathan Rose, Kevin Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays", *Proc. 27th ACM/IEEE Design Automation Conf.* pp. 613-619, 1990.

[7] E. A. Walkup, S. Hauck, G. Borriello, C. Ebeling, "Routing-Directed Placement for the Triptych FPGA," *1992 ACM First International Workshop on FPGAs* pp. 33-38, 1992.

[8] Sha, L., Dutton, R., "An Analytical Algorithm for Placement of Arbitarity Sized Rectangular Blocks", *Proc.* 22st pp 602-608, 1985.

[9] Burstein, M., Pelavin, R., "Hierarchical Wire Routing", *IEEE Trans. on CAD, Special Issue on Automatic Wire Routing, Ed. E.S. Kuh* CAD-2" October 1983, pp.223-224.

[10] Leiserson, C.E., Printer, R.Y., "Optimal Placement for River Routing", *Proc. 1981 CMU Conf. VLSI Systems and Computations*, pp 126-142, 1981.

[11] Sangiovanni-Vincentelli, A. Santomauro, M.,Read, J., "A New Gridless Channel Router: Yet Another Channel Router the Second(YACR-11)", *Proc. ICCAD*, 1984

[12] Sechen, C., Sangiovanni-Vincentelli, "The Timber-Wolf Placement and Routing Package", *IEEE Journal of Solid-State Circuits, Vol. SC-20, 2* April, 1985.

[13] M. Chrzanowska-Jeske, S. Goller, I. Schafer, "An Architecture-Driven Approach for the Fitting Problem in an Application- Specific EPLD", *IEEE International Symposium on Circuits and Systems, Proceedings Volume* 3 of 4 VLSI and Parallel Processing" p.1782, 1993

[14] M. Chrzanowska-Jeske, S. Goller, "Partitioning Approach to Find an Exact Solution to the Fitting Problem in an Application-Specific EPLD Device" *EURO-DAC CCH Hamburg, Germany* Sept. 20-24, 1993.

[15] Perkowski, M.A., Chrzanowska-Jeske, M., Coppola, A., Pierzchala, E., "An Exact Algorithm for the Technology Fitting Problem in the Application Specific State Machine Device," *ISCAS*, 1992.

[16] Steffen Goller. "Partitioning_Based Approach to The Fitting Problem In Special Architecture EPLDs", *Portland State University*, 1992.

[17] Yap, H.P., "Some Topics in Graph Theory", *Cambridge University Press, Cambridge* 1986, p.88.

[18] Cypress BIMOS CMOS DATA Book, 1992.

[19] Nam-Sung Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility", *Proc.* 28th pp. 248-251, San Francisico, CA, June 1991.

[20] R.J. Francis, J. Rose, Z. Vranesic, "Technology Mapping of Lookup Table Based FPGAs for Performance", *Proc. ICCAD* pp. 568-571, Nov. 1991.

[21] W.T. Tutte "Graph Theory", *Addison-Wesley Publishing Company* pp. 261, 1984